Prepared for
Andy Jarrett
Radix Publishing Limited

Prepared by
Syed Faraz Abrar
Can Bölük
Zellic

December 10, 2024

Radix
Blockchain Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Radix Publishing Limited from August 8th to December 2nd , 2024. During this engagement, Zellic reviewed Radix's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could an attacker cause a chain halt?
- Could an attacker use a vulnerability to gain access to unauthorized resources (particularly XRD)?
- Could an attacker abuse a vulnerability in the WASM instrumentation to avoid paying gas fees?
- Could an attacker bypass the badge authorization system to perform unauthorized actions on other blueprints?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

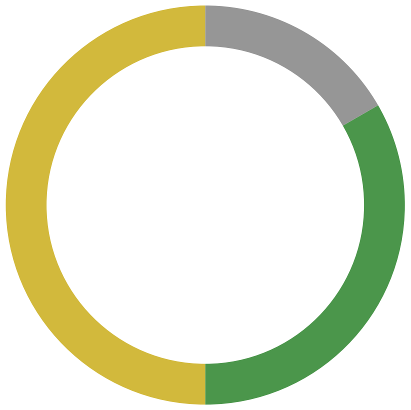
1.4. Results

During our assessment on the scoped Radix component, we discovered six findings. No critical issues were found. Three findings were of medium impact, two were of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Radix Publishing Limited in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	3
🟩 Low	2
⬜ Informational	1



2. Introduction

2.1. About Radix

Radix Publishing Limited contributed the following description of Radix:

Radix is a layer-1 smart contract protocol built for DeFi, providing a radically better user and developer experience.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the component.

Architecture risks. This encompasses potential hazards originating from the blueprint of a system, which involves its core validation mechanism and other architecturally significant constituents influencing the system's fundamental security attributes, presumptions, trust mode, and design.

Arithmetic issues. This includes but is not limited to integer overflows and underflows, floating-point associativity issues, loss of precision, and unfavorable integer rounding.

Implementation risks. This encompasses risks linked to translating a system's specification into practical code. Constructing a custom system involves developing intricate on-chain and off-chain elements while accommodating the idiosyncrasies and challenges presented by distinct programming languages, frameworks, and execution environments.

Availability. Denial-of-service attacks are another leading issue in custom systems. Issues including but not limited to unhandled panics, unbounded computations, and incorrect error handling can potentially lead to consensus failures.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped component itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Radix Component

Type	Rust
Platform	Radix

Target	radixdlt-scrypto
Repository	https://github.com/radixdlt/radixdlt-scrypto ↗
Version	`816f76fd06709a160a170b22944e636054323e71`
Programs	`Radix Engine`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of thirty-three person-weeks. The assessment was conducted by two consultants over the course of sixteen and a half calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Syed Faraz Abrar
Engineer
faith@zellic.io ↗

Can Bölük
Engineer
can.boluk@zellic.io ↗

2.5. Project Timeline

August 12, 2024	Kick-off call
August 8, 2024	Start of primary review period
December 2, 2024	End of primary review period

3. Detailed Findings

3.1. Missing subgroup check in BLS12-381 key and signature aggregation

Target	BLS12-381 Implementation		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	Medium

Description

The BLS12-381 key and signature-aggregation implementation in the VM skips critical subgroup membership checks for the first element in both public key and signature-aggregation operations. This affects the following components:

1. Public-key aggregation (radix-common/src/crypto/bls12381/public_key.rs)

```
pub fn aggregate(public_keys: &[Bls12381G1PublicKey]) -> Result<Self,
    ParseBlsPublicKeyError> {
    if !public_keys.is_empty() {
        let pk_first = public_keys[0].to_native_public_key()?;
        let mut agg_pk = AggregatePublicKey::from_public_key(&pk_first); // No
    validation
        for pk in public_keys.iter().skip(1) {
            agg_pk.add_public_key(&pk.to_native_public_key()?, true)?; //
    Validates subsequent keys
        }
        // ...
    }
    // ...
}
```

2. Signature aggregation (radix-common/src/crypto/bls12381/signature.rs)

```
pub fn aggregate(signatures: &[Bls12381G2Signature]) -> Result<Self,
    ParseBlsSignatureError> {
    if !signatures.is_empty() {
        let sig_first = signatures[0].to_native_signature()?;
        let mut agg_sig = AggregateSignature::from_signature(&sig_first); //
    No validation
        for sig in signatures.iter().skip(1) {
            agg_sig.add_signature(&sig.to_native_signature()?, true)?; //
```

```
     Validates subsequent signatures
         }
         // ...
     }
     // ...
}
```

The implementation omits subgroup checks for the first element in both cases, while correctly validating subsequent elements. This violates the security requirements specified in the BLS signature specification ↗ (draft-irtf-cfrg-bls-signature-04, Section 5.2).

Impact

The missing subgroup check on the first element during aggregation operations undermines the security guarantees of the BLS signature scheme.

Specifically, an attacker can provide a carefully crafted invalid point as the first element in an aggregation that, when combined with valid subsequent points, produces a signature that should be invalid but passes verification. The implementation allows signature malleability since points not in the correct subgroup can be used, violating the uniqueness properties required by the BLS signature scheme. The vulnerability enables related techniques from small subgroup attacks, potentially compromising the security assumptions around signature aggregation.

Recommendations

We recommend the following changes to the implementation:

1. Implement consistent subgroup validation for all elements.
2. Add explicit subgroup membership tests for all points before aggregation.
3. Add comprehensive test cases that specifically verify subgroup membership validation.

Remediation

This issue has been acknowledged by Radix Publishing Limited, and a fix was implemented in commit `184f2620` ↗.

3.2. Memory resource exhaustion via untracked buffers

Target	WASM API (Scrypto VM Implementation)		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

A vulnerability in the VM's buffer-management system allows an attacker to exceed memory limits through a combination of key-value store operations and raw syscalls. The issue stems from the following mechanisms.

1. First is the ability to store VM memory pages in a key-value store entry and retrieve them using raw syscalls, leading to an untracked memory allocation when the returned buffer entry is not immediately cleaned up.

```
fn materialize_duplicates(handle: u32, pages: &mut [u8]) {
    unsafe {
        // ... SBOR encoding setup ...
        wasm_api::kv_entry::kv_entry_write(handle, pages.as_ptr(), i + len);

        // Revert to initial value
        (pages.as_mut_ptr() as *mut u64).write_volatile(initial_value);
    }

    for _i in 0..31 { // buffer limit
        unsafe {
            std::mem::forget(wasm_api::kv_entry::kv_entry_read(handle));
        };
    }
}
```

2. Second is the ability to recursively create new call frames up to the depth limit that maintain separate buffer limits.

```
unsafe {
    wasm_api::blueprint::blueprint_call(
        package_address.as_bytes().as_ptr(),
        package_address.as_bytes().len(),
        blueprint_name.as_ptr(),
        blueprint_name.len(),
        function_name.as_ptr(),
        function_name.len(),
        args.as_ptr(),
        args.len(),
    );
}
```

The attack works by allocating the maximum available memory in the VM, storing this memory in a key-value store entry, creating duplicates through repeated reads while avoiding buffer cleanup, and recursively spawning new call frames to multiply the effect.

Impact

The vulnerability allows a malicious transaction to do the following:

- Consume approximately 480MB of validator memory (8 call frames x 32 buffer limit x 2MB transaction substate limit), which is a ~250x amplification of the intended memory limits per transaction
- Cause significant performance degradation with execution times of 5–10 seconds per transaction
- Create physical memory pressure on validator nodes through repeated physical page allocations and deallocations
- Potentially lead to out-of-memory conditions in validator processes if multiple transactions are executed concurrently

Recommendations

We recommend two potential approaches to resolve this issue:

1. **Buffer slot–limit reduction**

   The current limit of 32 buffer slots appears unnecessary as the `Buffer` type is only used by the private API. Most WASM APIs return only one buffer, and the application layer forces it to be consumed and cleaned up before returning. Consider reducing the buffer limit to two slots: one slot for general operations and one additional slot to support potential callback scenarios. This would reduce the maximum memory amplification from ~250x to ~16x.

2. **Alternative memory management**

   Consider exporting `malloc/free` from the guest environment, which would enable proper cost tracking in all scenarios. Memory allocations would be tracked and limited appropriately through the cost-unit system, and it would not require any copies in the host state.

The first approach (reducing buffer slots) is likely the simplest and most effective solution, as there are no known legitimate use cases requiring multiple buffer slots.

Remediation

This issue has been acknowledged by Radix Publishing Limited, and a fix was implemented in commit `cd5999b2` ↗.

3.3. Gas-accounting discrepancy for infinite loops

Target	Gas Metering (Scrypto VM Implementation)		
Category	Optimization	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

A `loop {}` instruction takes approximately 5.2 seconds to reach the `LimitExceeded` error, despite the system's baseline costing rule being designed for 1µs = 100 cost units (implying ~1 second execution for the default 100m limit).

The issue stems from the WASM instrumentation process generating inefficient gas-accounting code for simple loops:

```
pub fn fn() {
    loop {
      std::hint::black_box(());
    }
}
```

It generates WASM with the following structure:

```
loop  ;; label = @2
  i64.const 3529    // Gas cost per iteration
  call 11           // Gas accounting function call
  br 0 (;@2;)       // Branch back to start
end
```

The overhead of the gas-accounting function call (`call 11`) dominates the execution time since the original loop body is empty, making the actual runtime significantly exceed the intended cost limits.

Impact

Transactions can consume up to 5x more wall-clock time than intended by the gas limits, and there is also the potential for transaction-processing delays in blocks.

Recommendations

Several approaches can be taken to address this issue:

1. **Gas cost adjustments**

   Add a base cost for each gas metering operation. Adjust the gas-cost unit multiplier to better reflect actual execution costs.

2. **Loop optimization**

   Implement loop unrolling during instrumentation to reduce the frequency of gas-accounting calls. Here is an example transformation:

```
loop
  i64.const 14116  // Combined cost for multiple iterations
  call 11
  br 0
end
```

3. **Gas function optimization**

   Optimize the gas-accounting host function for the common case.

Remediation

This issue has been acknowledged by Radix Publishing Limited, and a fix was implemented in commit 86828df1 ↗.

3.4. Inefficient `Metadata` array validation sequence

Target	Metadata Native Blueprint		
Category	Optimization	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `Metadata` native blueprint's `create_with_data()` function performs validation in a suboptimal order that could lead to excessive resource consumption. Specifically, the `validate_metadata_value()` function validates array contents before their length is checked in `init_system_struct()`.

For array types like `MetadataValue::UrlArray`, this means expensive validation operations (e.g., URL validation that involves regex matching) are performed on all elements before the array size is verified to be within acceptable bounds:

```
pub fn instantiate_test_blueprint() -> Global<TestBlueprint> {
    // Create large vector of URLs to validate
    let metadata_config = metadata!(init {
        "test" => Self::test_create_vec(), // Vector of 15000+ URLs
        locked;
    });
    let metadata = Metadata::new_with_data(metadata_config.init);
    // ...
}

fn test_create_vec() -> Vec<UncheckedUrl> {
    let mut temp_vec: Vec<UncheckedUrl> = vec![];
    for i in 0..15000 {
        temp_vec.push(UncheckedUrl::of(
            format!("https://www.google.com/{i}")
        ));
    }
    temp_vec
}
```

The validation sequence flows as follows. First, `validate_metadata_value()` performs URL/origin validation:

```
pub fn validate_metadata_value(value: &MetadataValue) -> Result<(),
    MetadataValidationError> {
    match value {
        MetadataValue::UrlArray(urls) => {
            for url in urls {
                CheckedUrl::of(url.as_str())

    .ok_or(MetadataValidationError::InvalidURL(url.as_str().to_owned()))?;
            }
        }
        MetadataValue::OriginArray(origins) => {
            for origin in origins {
                CheckedOrigin::of(origin.as_str())

    .ok_or(MetadataValidationError::InvalidOrigin(origin.as_str().to_owned()))?;
            }
        }
        // ...
    }
    Ok(())
}
```

Only later in `init_system_struct()` is the array size checked against the 4,096 byte limit.

Preliminary testing shows that vector creation in blueprints is limited by WASM memory allocation (failing with `UnreachableCodeReached` trap when exceeded). Even within these memory bounds, validation can take 7–8 seconds for large vectors. Testing also shows that the transaction cost is ~4.69 XRD + 0.014 XRD for storage. The validation time is not properly reflected in the execution cost.

Impact

The validation-sequence inefficiency enables DOS vectors that

1. allow an attacker to cause disproportionate CPU consumption relative to transaction costs, and
2. Could potentially delay transaction processing when multiple such transactions are queued.

While the system's tip mechanism helps prioritize legitimate transactions during congestion, this issue provides a more cost-effective way to create that congestion in the first place.

Recommendations

We recommend the following changes to the implementation:

1. Reorder the validation sequence.

```
fn validate_metadata_value(value: &MetadataValue) -> Result<(),
    MetadataValidationError> {
    match value {
        MetadataValue::UrlArray(urls) => {
            // Check size before expensive validation
            let total_size = urls.iter()
            .fold(0, |acc, url| acc + url.as_str().len());
            if total_size > MAX_METADATA_SIZE {
               return Err(MetadataValidationError::ValueTooLarge);
               }
            // Proceed with validation knowing size is bounded
        }
        // ...
    }
}
```

2. Consider raising an error on the host side when memory allocation fails as opposed to returning -1 from `memory.grow` as per WASM specification since the memory allocator used by blueprints simply trap on this condition, leading to more confusing errors for developers.

Remediation

This issue has been acknowledged by Radix Publishing Limited, and a fix was implemented in commit `4dffd441` ↗.

3.5. Unbounded transaction reference validation

Target	Kernel Reference Validation		
Category	Optimization	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The kernel's `check_references()` function processes a transaction's reference addresses without enforcing any limit on their quantity. While transactions are bounded by a 1MB size limit, this still allows for approximately 34,000 references to be processed in a single transaction:

```
fn check_references(
    &mut self,
    callback: &mut M,
    references: &IndexSet<Reference>,
) -> Result<(IndexSet<GlobalAddress>, IndexSet<InternalAddress>),
    BootloadingError> {
    let mut global_addresses = indexset!();
    let mut direct_accesses = indexset!();

    // Unbounded iteration over references
    for reference in references.iter() {
        let node_id = &reference.0;

        if ALWAYS_VISIBLE_GLOBAL_NODES.contains(node_id) {
            continue;
        }
        // ... reference validation logic ...
    }

    Ok((global_addresses, direct_accesses))
}
```

Every reference requires the following:

1. Checking against always visible nodes
2. Validating the reference type
3. Reading substate information
4. Verifying the reference value

5. Adding to appropriate collections

Impact

Processing large numbers of references could increase block processing time as each reference requires substate reads and validation, add computational overhead during transaction validation, and potentially affect block-generation timing if transactions with many references are included.

The practical impact is low because the 1MB size limit provides an upper bound on the number of references that can be included in a single transaction.

Recommendations

We recommend enforcing a limit on the number of references that can be included in a transaction.

```
// @ radix-common/src/constants/transaction_execution.rs
pub const MAX_TRANSACTION_REFERENCES: usize = 1024;
```

```
fn check_references(
    &mut self,
    callback: &mut M,
    references: &IndexSet<Reference>,
) -> Result<(IndexSet<GlobalAddress>, IndexSet<InternalAddress>),
    BootloadingError> {
    if references.len() > MAX_TRANSACTION_REFERENCES {
        return Err(BootloadingError::TooManyReferences(references.len()));
    }
    // ... existing validation logic ...
}
```

Remediation

This issue has been acknowledged by Radix Publishing Limited, and a fix was implemented in commit ea2ff3c2 ↗.

3.6. Incomplete WebAssembly table-section validation

Target	WASM Limits (Scrypto Parser)		
Category	Code Maturity	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The WebAssembly module validator checks the initial size of tables against a configured maximum but does not check the table's specified maximum size limit,

```
pub fn enforce_table_limit(self, max_initial_table_size: u32) -> Result<Self,
    PrepareError> {
    let section = self.module.table_section()?;

    if let Some(section) = section {
        if section.len() > 1 {
            return
    Err(PrepareError::InvalidTable(InvalidTable::MoreThanOneTable));
        }

        if let Some(table) = section.get(0) {
            if table.ty.initial > max_initial_table_size {
                return Err(PrepareError::InvalidTable(
                    InvalidTable::InitialTableSizeLimitExceeded,
                ));
            }
            // Note: table.ty.maximum is not checked
        }
    }
    Ok(self)
}
```

while memory sections properly validate both initial and maximum size limits:

```
if let Some(max) = memory.maximum {
    if max > max_memory_size_in_pages.into() {
        return Err(PrepareError::InvalidMemory(
            InvalidMemory::MemorySizeLimitExceeded,
        ));
    }
}
```

Impact

Currently, this is not exploitable because the implementation uses a WebAssembly MVP target that does not support table growth, and relevant instructions like `table.grow` are disabled in the `wasm-parser` configuration as GC proposal support is not enabled.

Recommendations

To maintain consistency with memory validation and future-proof the codebase, we recommend the following validation for table sections:

```
pub fn enforce_table_limit(self, max_table_size: u32) -> Result<Self,
    PrepareError> {
    let section = self.module.table_section()?;

    if let Some(section) = section {
        if section.len() > 1 {
            return
    Err(PrepareError::InvalidTable(InvalidTable::MoreThanOneTable));
        }

        if let Some(table) = section.get(0) {
            // Check initial size
            if table.ty.initial > max_table_size {
                return Err(PrepareError::InvalidTable(
                    InvalidTable::InitialTableSizeLimitExceeded,
                ));
            }

            // Check maximum size if specified
            if let Some(max) = table.ty.maximum {
                if max > max_table_size {
                    return Err(PrepareError::InvalidTable(
                        InvalidTable::MaximumTableSizeLimitExceeded,
                    ));
                }
            }
        }
    }
    Ok(self)
}
```

Remediation

This issue has been acknowledged by Radix Publishing Limited.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Unique Characteristics of Radix

Throughout the course of this audit, we have noticed that Radix has many differences in relation to most other blockchains. Two aspects of Radix that we would like to note (and what makes Radix stand out from the rest of the blockchains) are as follows:

1. The resource and badge system
2. The security-through-ownership system

We briefly cover why these are innovative and novel and also discuss the security benefits of designing systems in this way.

Resource and badge system

Resources are to Radix what tokens are to Ethereum. The biggest difference is that resources are natively built into the Radix engine, as opposed to tokens, which are not native and must be implemented as smart contracts.

Why is this important? Having resources be a part of the engine itself allows the engine to maintain invariants as well as impose rules and restrictions regarding how resources are managed and used. In essence, this provides more control than a typical smart contract would have, and the control lies entirely with the blockchain developers themselves.

In Radix, there are two types of resources: fungible and nonfungible. When a new resource is created, it creates a new component out of a preexisting native blueprint. The native blueprint can be thought of as a smart contract from Ethereum, except that it is maintained by the Radix development team and can access the engine in ways that user-defined blueprints cannot (through an API that only they have access to). This in turn means that users do not have to write their own code for their own resources, which then reduces the attack surface, as new attack surfaces cannot be introduced due to users writing erroneous code.

Additionally, one of the most unique aspects of Radix is how the resource system ties into how access-control mechanisms are set up. In Radix, if you want to restrict access to certain functions or methods in blueprints or restrict access to certain resources, it is as easy as checking to see if the caller holds a certain nonfungible resource. In this context, the nonfungible resource would be called a "badge", and this badge is what allows callers to prove that they have the required permissions to perform some action.

Therefore, in order to set up access control, it is as simple as creating a new nonfungible resource and minting it out to necessary parties. It is a very clean and simple system, and it is incredibly difficult to introduce access-control issues through user error.

Another key point is that access control in Radix is the default. A good example of this is the native accounts blueprint. In Radix, every account is a component created from the native accounts blueprint, and so the actual logic within each account is restricted to what the native accounts blueprint implements. This ensures that any security measures (such as not allowing users access to accounts that they are not authorized to access) are implemented at a blockchain level. This makes it absolutely impossible for any implementation level issues to arise out of user error.

In contrast, to implement smart accounts on Ethereum, not only do user-defined smart contracts need to be used (see ERC-4337 for example), but off-chain entities must also be used to submit transactions on behalf of these smart accounts. Additionally, since smart contracts do not automatically contain access-control logic, all access-control must be implemented manually by the developers of the smart account system.

The complex nature of such implementations introduce bugs such as this one ↗ or this one ↗. It also makes the bugs very difficult to fix at times, as the fixes may need to be deployed to every smart account (although there are some modular designs which prevents the need to do this). In contrast, on Radix, a single blockchain upgrade is enough to fix any bug.

Having said that, the native accounts blueprint contains such simple logic, that we were not able to find any vulnerabilities in it during the audit. It is a testament to the phrase "simple is good".

Security-through-ownership system

While Radix has many sophisticated features like native resources, badges, and access-control systems, its fundamental security model is effectively simple, built on two key rules:

1. No dangling resources — they must either be persisted through ownership or destroyed.
2. Only the creator of a resource has the power to destroy it.

These two rules provide a strong security guarantee for the developers building on Radix with little mental overhead.

The power of this approach is evident when comparing implementations of common DeFi patterns. Consider flash loans: on Ethereum, they typically require complex reentrant call patterns where the lender calls into borrower code, which calls back to repay, with careful balance checking needed throughout. This complexity creates many opportunities for subtle bugs.

On Radix, the same functionality is achieved through simple ownership rules:

- The lender creates two buckets: borrowed funds and a "promise" NFT that only the lender can burn.
- The borrower receives both through a `.borrow()` call.
- To complete the loan, the borrower must call `.return()` with both repayment funds and the promise NFT.
- The lender verifies the repayment amount and burns the promise NFT.
- If the borrower fails to return the funds, the transaction will abort as the promise is left dangling.

This functionality is often a complex choreography of reentrant calls, but in Radix it is a straightforward exchange of resources, with clear intent and security guaranteed by the ownership rules themselves.

This pattern can be found throughout Radix at every level of implementation. Even core features like resource management rely on these ownership rules rather than complex invariants being checked. When depositing resources, the system simply `drop()`s the transient instance (bucket) and then increments the vault's balance. If an attacker tries to deposit a bucket holding a different kind of resource, the operation will abort as they (resource A) are not the creator (of bucket of resource B). Simple rules create strong security.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the component and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: Resource blueprints

Resources

In other blockchain systems, such as Ethereum, tokens or assets are instead known as "resources" in the Radix blockchain.

Resources must always be stored in containers. In Radix, there are two types of containers, and they are both implemented as native blueprints.

Resource managers

When a new resource is created, it is uniquely identified by its resource manager. The resource manager allows specific roles to access manager-like functionality. For example, it would allow someone with the `MINTER_ROLE` role permission to mint the resource to others.

Buckets

In Radix, buckets are temporary containers that are used to store resources during the lifetime of a transaction. If a transaction ends with buckets still containing resources, an error will occur and the transaction will be reverted.

Buckets can have resources taken out of or put into them by one party and transferred over to another party, who can then take the resources out of the bucket. This is the primary way resources are transferred between owners in Radix.

There are two types of buckets:

1. Fungible buckets — These are used to store fungible resources, similar to what ERC-20 tokens are on Ethereum.
2. Nonfungible buckets — These are used to store nonfungible resources, similar to what ERC-721 tokens are on Ethereum.

Vaults

Vaults behave the same as buckets in most cases. The main difference is that vaults are used to store resources permanently.

In order to move resources from one vault to another, they always have to go through a bucket.

Similar to buckets, the two types of vaults are fungible and nonfungible vaults.

Proofs

Proofs in Radix are created by both buckets and vaults. It is used to prove that the owner of the bucket or vault (i.e., the one who created the proof) has access to a specified amount of resources.

Fungible buckets

Fungible buckets are components that are created to hold fungible resources throughout the lifetime of a transaction. The owned component (which can be a blueprint or another component) is able to take resources out and put resources into this bucket.

Fungible buckets differentiate between liquid and locked amounts. Amounts are locked when a user creates a proof-of-amount from this bucket. The proof can be used to prove to others that this user has access to the locked amount of a resource. Any non-locked funds are considered liquid and can be transferred out of the bucket.

Method: `take()`

The method signature is `take(amount: Decimal, api: &mut Y)`.

This method is intended to be used to take resources out of the bucket. It returns a new bucket with the resources that were taken out.

This method simply calls `take_advanced(amount, WithdrawStrategy::Exact, api)`. The `amount` argument is user controlled.

Method: `take_advanced()`

The method signature is `take_advanced(amount: Decimal, withdraw_strategy: WithdrawStrategy, api: &mut Y)`.

This method contains the core logic behind taking resources out. The `WithdrawStrategy` is an enum:

```
pub enum WithdrawStrategy {
    Exact,
    Rounded(RoundingMode),
}
```

And `RoundingMode` is also an enum:

```
pub enum RoundingMode {
    /// The number is always rounded toward positive infinity, e.g. `3.1 ->
    4`, `-3.1 -> -3`.
    ToPositiveInfinity,
    /// The number is always rounded toward negative infinity, e.g. `3.1 ->
    3`, `-3.1 -> -4`.
    ToNegativeInfinity,
    /// The number is always rounded toward zero, e.g. `3.1 -> 3`, `-3.1 ->
    -3`.
    ToZero,
    /// The number is always rounded away from zero, e.g. `3.1 -> 4`, `-3.1 ->
    -4`.
    AwayFromZero,

    /// The number is rounded to the nearest, and when it is halfway between
    two others, it's rounded toward zero, e.g. `3.5 -> 3`, `-3.5 -> -3`.
    ToNearestMidpointTowardZero,
    /// The number is rounded to the nearest, and when it is halfway between
    two others, it's rounded away from zero, e.g. `3.5 -> 4`, `-3.5 -> -4`.
    ToNearestMidpointAwayFromZero,
    /// The number is rounded to the nearest, and when it is halfway between
    two others, it's rounded toward the nearest even number. Also known as
    "Bankers Rounding".
    ToNearestMidpointToEven,
}
```

When the `withdraw_strategy` is set to `WithdrawStrategy::Rounded`, the `amount` argument is rounded based on the `RoundingMode` that is set.

Afterwards, the `amount` itself is checked to ensure that it is nonnegative and divisible by `divisibility`, which is the amount of decimals this resource is using.

Then, the amount is taken out of this bucket and transferred over into a new bucket. This new bucket is then returned.

If the bucket does not have at least `amount` of the resource in it, the transaction is reverted. This only looks at the current liquid amount of the resource in the bucket and ignores locked resources.

Method: `put()`

The method signature is `put(bucket: Bucket, api: &mut Y)`.

This method is used to take fungible resources out of another bucket and insert into the bucket that this method is called on.

There are no direct checks to ensure that the `bucket` argument contains a bucket that has the same type of resource as the bucket this function is called on. However, when the `drop_fungible_bucket()` function is called to drop the other bucket (i.e., the `bucket` passed in), it ensures that

1. the other bucket does not currently have any locked funds, and
2. the other bucket was also created by the same blueprint.

The second check in particular prevents resources between different types of buckets from getting mixed up.

Function: `get_amount()`

The method signature is `get_amount<Y>(api: &mut Y)`.

This function simply adds up this bucket's liquid and locked amounts and returns it.

Function: `get_resource_address()`

The method signature is `get_resource_address<Y>(api: &mut Y)`.

This function returns the address of this bucket as a `ResourceAddress`. This can be used to determine the type of resource stored within this bucket.

Function: `create_proof_of_amount()`

The method signature is `create_proof_of_amount<Y>(amount: Decimal, api: &mut Y)`.

This function is used to create a proof-of-amount from this bucket. The proof can be used by the caller to prove that they have access to a certain amount of resources from this bucket.

When the proof is created, the `amount` argument is used to determine the amount of resources to lock for this proof.

This `amount` cannot be negative and must also be divisible by the `divisibility` of this resource.

Method: `create_proof_of_all()`

The method signature is `create_proof_of_all<Y>(api: &mut Y)`.

This function calls `create_proof_of_amount()`, passing in the entire amount of the bucket as the first argument. See the above subsection for more details.

Nonfungible buckets

Nonfungible buckets are components that are created to hold nonfungible resources throughout the lifetime of a transaction. The owned component (which can be a blueprint or another component) is able to take resources out and put resources into this bucket.

Nonfungible buckets also differentiate between liquid and locked resources. Specified nonfungible IDs are locked when a user creates a proof from this bucket. The proof can be used to prove to others that this user has access to the locked nonfungible IDs. Any non-locked IDs are considered liquid and can be transferred out of the bucket.

Below is a list of all the functions and methods a nonfungible bucket exposes.

Method: `take()`

The method signature is `take(amount: Decimal, api: &mut Y)`.

This method is intended to be used to take the specified `amount` of nonfungible resources from this bucket. It returns a new bucket with the resources that were taken out.

This method simply calls `take_advanced(amount, WithdrawStrategy::Exact, api)`. The `amount` argument is user controlled.

Method: `take_advanced()`

The method signature is `take_advanced(amount: Decimal, withdraw_strategy: WithdrawStrategy, api: &mut Y)`.

This method contains the core logic behind taking resources out. The `withdraw_strategy` works the same way as it did with the fungible bucket.

The `amount` itself is ensured to be positive and able to fit within a `u32`. There is no `divisibility` to check for nonfungible resources.

Since nonfungible resources are not fungible, the `amount` is used to determine how many resources to remove from this bucket. The resources are removed in ascending order (that is, ID 1 is removed before ID 2 and onwards).

If the bucket does not contain enough liquid resources to take out, an error is returned and the transaction is reverted.

Method: `take_non_fungibles()`

The method signature is `take_non_fungibles<Y>(ids: &IndexSet<NonFungibleLocalId>, api: &mut Y)`.

This version of the `take_*()` function is used to take out specific nonfungible resources from this bucket by ID. The user passes in an index set of the IDs they want to take out.

If any of the IDs do not exist in this bucket or are currently locked, an error is returned and the transaction is reverted.

Method: `put()`

The method signature is `put(bucket: Bucket, api: &mut Y)`.

This method is used to take nonfungible resources out of another bucket and insert into the bucket that this method is called on.

There are no direct checks to ensure that the `bucket` argument contains a bucket that has the same type of resource as the bucket this function is called on. However, when the `drop_non_fungible_bucket()` function is called to drop the other bucket (i.e., the `bucket` passed in), it ensures that

1. the other bucket does not currently have any locked funds, and
2. the other bucket was also created by the same blueprint.

The second check in particular prevents resources between different types of buckets from getting mixed up.

Function: `get_non_fungible_local_ids()`

The method signature is `get_non_fungible_local_ids<Y>(api: &mut Y)`.

This function creates a new index set containing all the nonfungible resources this bucket contains, both liquid and locked. The index set is returned to the caller.

Function: `contains_non_fungible()`

The method signature is `get_non_fungible_local_ids<Y>(id: NonFungibleLocalId, api: &mut Y)`.

This function is used to check whether the nonfungible resource ID specified by the `id` argument exists in this bucket. It checks both liquid and locked resources.

Function: `get_amount()`

The method signature is `get_amount<Y>(api: &mut Y)`.

This function simply adds up this bucket's liquid and locked amounts and returns it.

Method: `get_resource_address()`

The method signature is `get_resource_address<Y>(api: &mut Y)`.

This method returns the address of this bucket as a `ResourceAddress`. This can be used to determine the type of resource stored within this bucket.

Function: `create_proof_of_non_fungibles()`

The method signature is `create_proof_of_non_fungibles<Y>(ids: IndexSet<NonFungibleLocalId>, api: &mut Y)`.

This method is used to create a proof-of-nonfungibles from this bucket. The proof can be used by the caller to prove that they have access to the set of nonfungible resources specified in the `ids` argument. These resources are locked for this proof.

If any of the resources specified in the `ids` argument do not exist in this bucket or are already locked, an error is returned and the transaction is reverted.

Method: `get_resource_address()create_proof_of_all()`

The method signature is `create_proof_of_all<Y>(api: &mut Y)`.

This method calls `create_proof_of_non_fungibles()`, passing in an index set that contains all the nonfungible resources in this bucket. See the above subsection for more details.

Fungible vaults

Fungible vaults are very similar to fungible buckets. The main differences are the following features:

1. They cannot have resources put into or taken out of them if they are frozen. Freezing and unfreezing vaults are access controlled.
2. XRD vaults in particular can be used to lock fees for transactions.
3. They support recall functionality. This is especially useful for "renting" resources to others. The special recaller role has the ability to recall resources from any vault of this type.

4. They can burn resources, deleting them from existence forever.

Below is a list of all the functions that are unique to vaults. Please see section 5.1. ↗ for the rest of the functions, as they are the same.

Method: `lock_fee()`

The method signature is `lock_fee(amount: Decimal, contingent: bool, api: &mut Y)`.

This method is intended to be used only on XRD vaults. All other vaults will return an error if this method is called on them.

The method performs the same checks on the `amount` as the `take_advanced()` method does. It then ensures that the COSTING module is enabled, to ensure that fees cannot be locked when the module is disabled.

The method then goes through the same flow as `take_advanced()` in order to remove `amount` resources from this vault. Finally, it calls the Radix Engine's `lock_fee()` function, which handles using this fee to pay for the transaction.

More information on how the Radix Engine handles locking the fee can be found in section 5.2. ↗.

Method: `recall()`

The method signature is `recall(amount: Decimal, api: &mut Y)`.

This method is used to recall resources. It can only be called by the `RECALLER_ROLE`. Additionally, the `VaultRecall` feature must be enabled when the vault is created.

Otherwise, it performs the exact same flow of actions as the `take_advanced()` function in order to remove resources from the vault.

Method: `freeze()`

The method signature is `freeze(to_freeze: VaultFreezeFlags, api: &mut Y)`.

This method is used to freeze the vault. The `to_freeze` argument is of type `VaultFreezeFlags`:

```
bitflags! {
    #[cfg_attr(feature = "fuzzing", derive(Arbitrary))]
    #[derive(Sbor)]
    pub struct VaultFreezeFlags: u32 {
        const WITHDRAW = 0b00000001;
        const DEPOSIT = 0b00000010;
        const BURN = 0b00000100;
    }
}
```

Effectively, the `FREEZER_ROLE` can freeze withdrawals, deposits, or burns (and any combination of the three). Additionally, the `VaultFreezable` feature must be enabled when the vault is created.

Method: `unfreeze()`

The method signature is `unfreeze(to_freeze: VaultFreezeFlags, api: &mut Y)`.

This method is used to unfreeze the vault. In this case, the `FREEZER_ROLE` can unfreeze withdrawals, deposits, or burns (and any combination of the three). Additionally, the `VaultFreezable` feature must be enabled when the vault is created.

Nonfungible vaults

Nonfungible vaults are very similar to nonfungible buckets. The main differences are the following features:

1. Nonfungible vaults cannot have resources put into or taken out of them if they are frozen. Freezing and unfreezing vaults are access controlled.
2. They support recall functionality. This is especially useful for "renting" resources to others. The special recaller role has the ability to recall resources from any vault of this type.
3. They can burn resources, deleting them from existence forever.

The functions that are unique to nonfungible vaults have the same behavior as the functions unique to fungible vaults:

1. Recalling, freezing, and unfreezing vaults are protected and thus can only be called by the `RECALLER_ROLE` and `FREEZER_ROLE`, respectively.
2. The `VaultRecall` and `VaultFreezable` features must be enabled to use the aforementioned features.

Fungible resource managers

Fungible resource managers are created for fungible resource types. Each type of resource will have its own resource manager that uniquely identifies this type.

The resource managers have methods that allow the owner to perform admin-like actions. This includes minting resources, burning resources, updating the resource metadata, and changing permissions (such as changing whether the resource is mintable, burnable, etc.).

Below is a list of all the functions and methods a fungible resource manager exposes.

Function: `create()`

The function signature is `create(owner_role: OwnerRole, track_total_supply: bool, divisibility: u8, resource_roles: FungibleResourceRoles, metadata: ModuleConfig<MetadataInit>, address_reservation: Option<GlobalAddressReservation>, api: &mut Y)`.

This function is part of the resource package and is intended to be used to create a new resource manager for a new fungible resource.

The `owner_role` argument is of type `OwnerRole`, which is the following enum:

```
pub enum OwnerRole {
    /// No owner role
    None,
    /// Rule protected Owner role which may not be updated
    Fixed(AccessRule),
    /// Rule protected Owner role which may only be updated by the owner
    themself
    Updatable(AccessRule),
}
```

The two types of owner roles (aside from the `None` type) determine whether the owner role can be updated later on or whether it is fixed forever.

The first thing the function does is create a node for the new resource manager. In creating this node, it ensures that the `divisibility` specified is valid (which means it is between 0 and 18 inclusive). When doing this, it stores a few fields into this node:

1. The `divisibility` is stored in its own field.
2. A `TotalSupply` field is created if `track_total_supply` is `true`.

The roles specified in `resource_roles` are also stored as part of the node. This is so that the node can later ensure that any callers of role-protected functions have the required roles to call them. These roles are as follows:

```
pub struct FungibleResourceRoles {
    pub mint_roles: Option<MintRoles<RoleDefinition>>,
    pub burn_roles: Option<BurnRoles<RoleDefinition>>,
    pub freeze_roles: Option<FreezeRoles<RoleDefinition>>,
    pub recall_roles: Option<RecallRoles<RoleDefinition>>,
    pub withdraw_roles: Option<WithdrawRoles<RoleDefinition>>,
    pub deposit_roles: Option<DepositRoles<RoleDefinition>>,
}
```

Once the node is created, if the user provided an address reservation in `address_reservation`, then the function calls into the kernel to create this address reservation. This is a way for the caller to know and decide the address of this resource ahead of time.

Finally, the node is globalized, which then allows the newly created resource-manager blueprint to be accessible by other blueprints in the global context. The resource manager's address is returned to the caller.

Function: `create_with_initial_supply()`

The function signature is `create_with_initial_supply(owner_role: OwnerRole, track_total_supply: bool, divisibility: u8, initial_supply: Decimal, resource_roles: FungibleResourceRoles, metadata: ModuleConfig<MetadataInit>, address_reservation: Option<GlobalAddressReservation>, api: &mut Y)`.

This function has the same logic as the `create()` function but also lets the caller pass in an `initial_supply`. This value is then checked to ensure it is divisible by `divisibility`, not negative, and not greater than `2**152`.

Additionally, this function returns a bucket containing the minted resources (i.e., the initial supply) along with the address of the newly created resource manager.

Method: `mint()`

The method signature is `mint(amount: Decimal, api: &mut Y)`.

This method is only callable by callers that have access to the `MINTER_ROLE`.

This method first ensures that this resource is mintable (meaning it has the `Mint` feature enabled). It then ensures that the `amount` argument is divisible by this resource's divisibility and that the amount is positive and not greater than the maximum mint amount of `2**152`.

If all of these checks pass, the method creates a new bucket with the minted resources. It also updates the resource's `TotalSupply` field by the minted amount if the `TrackTotalSupply` feature was set when creating this resource.

Method: `burn()`

The method signature is `burn(bucket: Bucket, api: &mut Y)`.

This method is only callable by callers that have access to the `BURNER_ROLE`.

This method first ensures that this resource is burnable (meaning it has the `Burn` feature enabled).

It then drops the bucket, which has the same flow as when a fungible bucket's `put()` method is called (which puts the resources from another bucket into the bucket that it is called on). The `drop_fungible_bucket()` method is used to do this, which ensures that that resource in the bucket originated from this resource manager.

Once the bucket is dropped, the burn flow is complete. Optionally, if this resource's `TrackTotalSupply` feature is enabled, the `TotalSupply` field is updated to account for the burned resources.

Method: `package_burn()`

The method signature is `package_burn(bucket: Bucket, api: &mut Y)`.

This method is the exact same as the `burn()` method, except it can only be called within the package. It allows the burning of tokens by vaults.

Method: `drop_empty_bucket()`

The method signature is `drop_empty_bucket(bucket: Bucket, api: &mut Y)`.

This method is used to drop empty buckets of this resource. It is primarily used to get rid of empty buckets in transactions (because otherwise the transaction will revert if there are buckets still in the worktop).

There are checks here to ensure that the bucket does not have any locked or liquid resources in it.

Method: `create_empty_bucket()`

The function signature is `create_empty_bucket(api: &mut Y)`.

This function creates an empty bucket for this resource and returns it.

Function: `create_empty_vault()`

The function signature is `create_empty_vault(api: &mut Y)`.

This function creates an empty vault for this resource and returns it.

Method: `get_resource_type()`

The method signature is `get_resource_type(api: &mut Y)`.

This method returns the resource type of this resource.

```
pub enum ResourceType {
    /// Represents a fungible resource
    Fungible { divisibility: u8 },

    /// Represents a non-fungible resource
    NonFungible { id_type: NonFungibleIdType },
}
```

Method: `get_total_supply()`

The method signature is `get_total_supply(api: &mut Y)`.

This method returns the current total supply of this resource.

Method: `amount_for_withdrawal()`

The method signature is `amount_for_withdrawal(api: &mut Y, amount: Decimal, withdraw_strategy: WithdrawStrategy)`.

This method returns the amount of tokens that would actually be withdrawn out of a vault or bucket for a specified `amount` and `withdraw_strategy`.

Nonfungible resource managers

Nonfungible resource managers are created for nonfungible resource types. Each type of resource will have its own resource manager that uniquely identifies this type.

The resource managers have methods that allow the owner to perform admin-like actions. This includes minting resources, burning resources, updating the resource metadata, and changing permissions (such as changing whether the resource is mintable, burnable, etc.).

The functions and methods in a nonfungible resource manager are very similar to the ones in the fungible resource manager. The differences are the same between fungible and nonfungible buckets (for example, minting is done through an index map of nonfungible IDs, rather than by a specified `amount`).

There is an additional mint method for `RUID`-type nonfungible resources, but aside from being a different type of nonfungible resource, it is the same.

Validator

A validator component is created when the `create_validator()` method is called on the consensus manager component. It is a component of the validator native blueprint.

It contains methods that allow validators to perform special actions, such as staking to themselves, registering to become a live validator, modifying validator-specific configuration (such as the fee factor), and more.

It also contains public methods callable by anyone. These methods are primarily used to add stake as a delegator, claim staking rewards, and so on.

We cover the most crucial publicly accessible methods here. We have, however, audited the validator-only methods and ensured that they behave as specified.

Method: `stake()`

The method signature is `stake(xrd_bucket: Bucket, api: &mut Y)`.

This method is used by delegators to stake their XRD to the validator that this method is called on.

The method first ensures that validators are accepting delegated stake, as that is a configurable option (i.e., validators can choose not to allow others to stake onto them).

The method then takes the amount of XRD in the `xrd_bucket` argument, inserts it into the validator's staked XRD vault, and then returns a corresponding amount of stake units in a bucket. The stake units signify that this delegator has an amount of stake on this validator.

The consensus-manager component's state is then updated with the new details of the validator (such as the updated total stake).

Method: `unstake()`

The method signature is `unstake(stake_unit_bucket: Bucket, api: &mut Y)`.

This method is used by delegators to unstake their XRD from the validator that this method is called on.

The `stake_unit_bucket` argument is the bucket of stake unit resources that are returned when the `stake()` method is called. A nonfungible token is returned to the user, and the corresponding XRD amount is locked into a specific vault. This XRD amount can be claimed later on by the delegator after `num_unstake_epochs` epochs have passed. The nonfungible token must be used to claim the XRD.

The consensus-manager component's state is then also updated.

Method: `claim()`

The method signature is `claim_xrd(bucket: Bucket, api: &mut Y)`.

This method is used by delegators to claim their unstaked XRD after `num_unstake_epochs` have passed. The `bucket` argument contains the unstake nonfungible token that is returned when the `unstake()` method is called.

The nonfungible token is burned, and there are checks to ensure that the caller is not attempting to claim nonexisting unstaked funds or unstaked funds that have not been unstaked for a sufficient amount of time.

Account locker

The account-locker native blueprint is intended to be used to lock resources into instantiated locker components. These resources can then later be claimed when certain conditions are met.

Typically, this would be useful for airdrops or bridges between Radix and other blockchain networks.

Users with the `STORER_ROLE` are allowed to store resources into the locker. Along with locking the resources, they are also allowed to specify a claimant (or list of claimants) for the stored resources. The claimant accounts have a nonfungible resource deposited into them that allows them to claim the resource afterwards. This is how access control is achieved, as anyone without the nonfungible resource badge cannot claim the resources.

Additionally, there is also a `RECOVERER_ROLE` that can recover any stored resources without needing to have access to the corresponding claimer nonfungible resource badges.

Worktop

The worktop is its own native blueprint. Each transaction has a worktop where resources are placed during the transaction.

The worktop cannot be accessed by user-created blueprints directly. It is created by the transaction-processor blueprint and used there.

Because the worktop is not directly accessible by user code but by instructions that are specified through the transaction manifest, we will not be covering the functions in detail here.

However, it is important to note that the worktop code does ensure that that buckets added to it are not empty and that amounts of resources taken out of it do exist in the worktop. This prevents users from submitting transactions that attempt to gain access to resources that do not exist.

Account

Unlike other blockchains, accounts in Radix are components. These components are instantiated from the native-account blueprint.

Each account component contains an owner, and the component is able to manage resources, lock XRD for transaction payments, and more.

Additionally, each account also has a special role called the Securify role, which is able to modify the account's authorization model.

It is also important to note that the identity native blueprint is a simpler version of the native-account blueprint, so the threat model for that would be a subset of the threat model presented here.

Let us look into the most security-critical functions and methods that the native-account blueprint exposes.

Function: `on_virtualize()`

The function signature is `on_virtualize(input: OnVirtualizeInput, api: &mut Y)`.

This function is called when a Radix address is interacted with (say, a deposit is sent to it) while there is no instantiated account component on that address. The `OnVirtualizeInput` argument type is as follows:

```
pub struct OnVirtualizeInput {
    pub variant_id: u8,
    pub rid: [u8; NodeId::RID_LENGTH],
    pub address_reservation: GlobalAddressReservation,
}
```

The `variant_id` is used to determine whether the `rid` is a SECP256K1 public key or an ED25519 public key. This function then calls `create_virtual()`, which is used to create and instantiate a native-account component at the address specified by `address_reservation` (which is the address that was interacted with).

When creating the account, an owner badge is created. The owner badge is a nonfungible resource that has the byte format of `[entity_type, public_key_hash]`, where `entity_type` determines whether the account is using a SECP256K1 public key or an ED25519 public key.

The owner badge is put into the account's metadata, along with the public key hash.

Radix uses an off-chain authentication method known as ROLA (Radix Off Ledger Authentication). A user can use this to prove that that they own the private key to the public key for a specific virtual account.

Function: `create()`

The function signature is `create(api: &mut Y)`.

This method is used to create an account component. It follows the same flow as `create_virtual()`, described in the threat model for `on_virtualize()` in the subsection above.

Function: `create_advanced()`

The function signature is `create_advanced(owner_role: OwnerRole, address_reservation: Option<GlobalAddressReservation>, api: &mut Y)`.

This has the same flow as the `create()` function, except that it also allows to specify a fixed or updatable owner role as well as a predefined address reservation for the account component.

Method: `lock_fee()`

The method signature is `lock_fee(amount: Decimal, api: &mut Y)`.

This method fetches the account's XRD vault, and locks a fee from it. It is intended to be used to pay for transaction fees. The core details behind transaction fees are covered in the transaction-flow threat model, which you can find in section 5.2. ↗.

This method is only callable by the owner of this account.

Method: `lock_contingent_fee()`

The method signature is `lock_contingent_fee(amount: Decimal, api: &mut Y)`.

This method does the same thing as `lock_fee()`, except that it locks a contingent fee. Contingent fees are fees that are only used if the transaction completes successfully.

This method is only callable by the owner of this account.

Method: `add_authorized_depositor()`

The method signature is `add_authorized_depositor(badge: ResourceOrNonFungible, api: &mut Y)`.

This method is used to add a nonfungible resource (which acts as a badge in this case) as an authorized depositor. This then allows anyone holding that nonfungible resource to deposit resources into this account.

This method is only callable by the owner of this account.

Method: `try_deposit_or_refund()`

The method signature is `try_deposit_or_refund(bucket: Bucket, authorized_depositor_badge: Option<ResourceOrNonFungible>, api: &mut Y)`.

This method is callable by anyone on every account. It is intended to be used to deposit resources into an account.

The resources that are deposited come from the `bucket` argument. If this account is not accepting deposits of the resource within the `bucket`, then the `authorized_depositor_badge` is checked to ensure that the depositor has been authorized to deposit into this account.

If the depositor is not authorized, then the `bucket` is refunded to the caller (i.e., it is returned by this function).

Method: `try_deposit_or_abort()`

The method signature is `try_deposit_or_abort(bucket: Bucket, authorized_depositor_badge: Option<ResourceOrNonFungible>, api: &mut Y)`.

This method calls `try_deposit_or_refund()`. The difference is that if the deposit fails, then the entire transaction is reverted.

Liquidity pools

The Radix engine has three native blueprints for liquidity pools. These are the one, two, and multiresource pools.

The functionality of all three pools are mostly the same. Every blueprint has an `instantiate()` function, which is intended to be used to create a pool component. The pool component then contains methods that allow the pool-manager role to contribute resources to the pool. The pool then returns a resource bucket that contains the equivalent of LP tokens. These LP tokens can then be used to redeem the pool resources back at a later time.

Typically, the pool-manager role would be assigned to an exchange blueprint. The exchange blueprint would allow users to swap resources within the pool or even between multiple pools.

There are also publicly accessible redeem functions that can be used to redeem pool resources using buckets of LP resources. These are intended to be used by normal users, while the protected deposit and withdraw methods are intended to be used by the pool manager.

Access controller

The access-controller native blueprint is used to instantiate access-controller components. These components are intended to hold badges, which then allow authorized callers to generate proofs of these badges. This in turn lets the authorized callers use the badges for authentication elsewhere.

Every method of the access-controller component is protected; that is, only authorized roles are able to call these methods. We have extensively audited the code here to ensure that the access-controller component behaves as intended.

The only method that is callable by anyone is the `timed_confirm_recovery()` method. The recovery process is intended to be used when the role definitions within the access controller need to be updated (one situation where this might be necessary is if one of the roles is compromised).

In order to do this, the primary or recovery role owner must initiate the recovery process. Once the recovery proposal has gone through, if a certain amount of time has passed, the recovery proposal can be confirmed by anyone. To confirm this recovery, the `timed_confirm_recovery()` method must be called.

The method signature is as follows, where the `rule_set` contains the new role definitions for the primary, recovery, and confirmation roles:

```
timed_confirm_recovery<Y>(
        AccessControllerTimedConfirmRecoveryInput {
            rule_set,
            timed_recovery_delay_in_minutes,
        }: AccessControllerTimedConfirmRecoveryInput,
        api: &mut Y,
    )
```

Consensus manager

The consensus-manager blueprint is responsible for handling consensus between validator nodes.

A consensus-manager component was created at genesis by the system bootstrap transaction. New epochs are created under the following conditions:

1. If the current round in this epoch is below `min_round_count`, the current epoch stays.
2. If the current round in this epoch is above `max_round_count`, a new epoch is created.
3. If neither conditions 1 nor 2 are the case, then a new epoch is created if more than `target_duration_millis` time has passed in the current epoch.

The consensus manager contains getter functions that can be used to deterministically fetch the current time in the context of the blockchain. This is done by fetching the timestamp that the latest round was started on.

The two most important functions are the `create_validator()` and `next_round()` methods, which handle creating new validators and advancing the consensus state to the next round, respectively. The `next_round()` method in particular is only callable through a system transaction, and thus we will not cover the details of this function here. However, we have not found any issues in the method logic after auditing this method.

Method: `create_validator()`

The method signature is `create_validator(key: Secp256k1PublicKey, fee_factor: Decimal, xrd_payment: Bucket, api: &mut Y)`.

This method can be called by anyone to create a new validator. The `key` argument is used as the public key for the newly created validator.

The `xrd_payment` bucket must contain enough XRD in it to pay for the validator-creation cost, which is a configurable field in the consensus-manager component. The XRD in the bucket is burned.

The `fee_factor` argument is used to determine the amount of the emissions that the validator owner keeps. This is important in the context of delegators staking their tokens to a validator.

This function creates a component of the validator native blueprint and returns the address.

Fungible and nonfungible proofs

Both fungible and nonfungible proofs are created whenever the corresponding proof-creation function is called on a fungible bucket or vault.

The created proofs can either be cloned or dropped. Additionally, there are some functions that are called when this proof is moved between call frames or when the node that owns this proof is dropped.

Method: `clone()`

The method signature is `clone(api: &mut Y)`.

This method is used to create a clone of the proof that this method is called on.

When the proof is cloned, both the `Moveable` and `ProofRefs` fields are copied over. The `Moveable` field contains a `restricted` variable that determines whether this proof is able to be moved between call frames. The `ProofRefs` field contains the total amount of resources that are locked into this proof as well as the containers (which can be buckets or vaults) that the corresponding amount of resources have been locked from.

Additionally, whenever a proof is cloned, the amount of resources that it has locked is duplicated. This means that if a proof contains 100 locked resources, cloning it then means that the cloned proof will also contain 100 locked resources, meaning the total amount of locked resources is now 200.

Method: `drop()`

The method signature is `drop(api: &mut Y)`.

This method is used to drop the proof that it is called on.

Method: `on_drop()`

The method signature is `on_drop(api: &mut Y)`.

This method is not externally callable. It is called when the node that owns this proof is dropped.

It ensures that the resources locked by this proof are unlocked when the owning node is dropped. Otherwise, it would lead to resources staying locked forever with no way to unlock them, as without the owning node, there would be no way to use this proof.

Method: `on_move()`

The method signature is `on_move(is_moving_down: bool, is_to_barrier: bool, destination_blueprint_id: Option<BlueprintId> api: &mut Y)`.

This method is not externally callable. It is called when this proof is being moved between call frames.

If this proof is being moved down into a child call frame, it checks that the proof is being moved to either an auth zone or another globalized component. If so, it checks to ensure that the `Moveable` field of the proof states that the proof is unrestricted and able to be moved. Otherwise, the transaction is reverted.

If the proof is able to be moved and it is being moved to another globalized component, then the `Moveable` field is modified and made to be restricted so that the proof cannot be moved any further.

Authorization zone

The authorization zone holds fungible and nonfungible proofs that are then automatically used to perform authorization checks for function/method calls.

Additionally, the authorization zone also contains functionality that allows fungible and nonfungible proofs to be constructed out of the proofs that are currently in the authorization zone.

Method: `push()`

The method signature is `push(proof: Proof, api: &mut Y)`.

This method is used to push a proof to the authorization zone. There are no checks performed for this; the proofs are simply pushed to the `AuthZone` field of this blueprint, which contains a vector of proofs.

Method: `pop()`

The method signature is `pop(api: &mut Y)`.

This method is used to pop a proof from the authorization zone. There are no checks performed for this. If there are no proofs in the authorization zone, then `None` is returned.

Method: `create_proof_of_amount()`

The method signature is `create_proof_of_amount(resource_address: ResourceAddress, amount: Decimal, api: &mut Y)`.

This method composes a new proof out of all the proofs in the authorization zone whose resource address matches the specified `resource_address`.

It ensures that the `amount` passed in can be satisfied by the proofs in the authorization zone.

A new proof is created and returned to the caller.

Method: `create_proof_of_all()`

The method signature is `create_proof_of_all(resource_address: ResourceAddress, api: &mut Y)`.

This method is the same as `create_proof_of_amount()`, except it creates a proof with the maximum amount of the specified resource in the authorization zone.

Method: `create_proof_of_non_fungibles()`

The method signature is `create_proof_of_all(resource_address: ResourceAddress, ids: IndexSet<NonFungibleLocalId>, api: &mut Y)`.

This method is the same as the `create_proof_of_amount()`, except it only works for nonfungible proofs and ensures that all nonfungible IDs specified in the `ids` index set exist in the authorization zone's list of proofs.

Method: `drop_proofs()`

The method signature is `drop_proofs(api: &mut Y)`.

This method drops all proofs from this authorization zone.

Method: `drop_signature_proofs()`

The method signature is `drop_signature_proofs(api: &mut Y)`.

This method removes all signature proofs from this authorization zone.

Method: `drop_regular_proofs()`

The method signature is `drop_regular_proofs(api: &mut Y)`.

This method drops all non-signature proofs from this authorization zone.

Method: `drain()`

The method signature is `drain(api: &mut Y)`.

This method removes all non-signature proofs from this authorization zone and returns them to the caller.

Method: `assert_access_rule()`

The method signature is `assert_access_rule(access_rule: AccessRule, api: &mut Y)`.

This method is used to ensure that the proofs in the current authorization zone can be used to authorize the specified `access_rule`.

The `access_rule` specifies what proof needs to exist in this authorization zone. In this case, there are five types of proofs:

```
pub enum ProofRule {
    Require(ResourceOrNonFungible),
    AmountOf(Decimal, ResourceAddress),
    CountOf(u8, Vec<ResourceOrNonFungible>),
    AllOf(Vec<ResourceOrNonFungible>),
    AnyOf(Vec<ResourceOrNonFungible>),
}
```

The `Require(ResourceOrNonFungible)` rule is used to ensure that the authorization zone contains a specific fungible resource or a specific nonfungible resource ID. The `AmountOf(Decimal, ResourceAddress)` rule is used to ensure that a specific amount of the resource specified by the `ResourceAddress` exists in this authorization zone; `CountOf(...)` is used to ensure that at least a specified amount of resources (which are specified in the vector) exist in this authorization zone; `AllOf(...)` is used to ensure that all of the resources specified in the vector exist in this authorization zone, and finally `AnyOf(...)` is used to ensure that at least one of the resources specified in the vector exist in this authorization zone.

Package

The package native blueprint contains functions that allow users of Radix to publish their own blueprints.

The functions in this blueprint are not callable directly. They must be called through either the transaction processor (when publishing a WASM blueprint, which is a user-defined blueprint) or through a system transaction (when publishing a native blueprint).

When publishing a user-defined WASM blueprint, there are rigorous checks to ensure that it adheres to certain criteria, which ensures that it is compatible with the Radix engine. Additionally, the WASM code is instrumented with instruction and stack-metering code that ensures that gas is appropriately consumed when executing WASM instructions and using the WASM VM stack.

We have extensively audited the user-defined blueprint publishing flow, and there was one gas-consumption–related finding that was found as a result of this. See finding 3.3. ↗.

5.2. Module: VM and runtime

Flow: Linking of guest code

The processing of guest imports starts before the virtual machine is instantiated at the validation and instrumentation stage, implemented within `ScryptoV1WasmValidator::validate`.

```rust
impl ScryptoV1WasmValidator {
    pub fn validate<'a, I: Iterator<Item = &'a BlueprintDefinitionInit>>(
        &self,
        code: &[u8],
        blueprints: I,
    ) -> Result<(Vec<u8>, Vec<String>), PrepareError> {
        WasmModule::init(code)?
            .enforce_no_start_function()?
            .enforce_import_constraints(self.version)? // <--- Here
            .enforce_export_names()?
    .enforce_memory_limit_and_inject_max(self.max_memory_size_in_pages)?
            .enforce_table_limit(self.max_initial_table_size)?
            .enforce_br_table_limit(self.max_number_of_br_table_targets)?
            .enforce_function_limit(
                self.max_number_of_functions,
                self.max_number_of_function_params,
                self.max_number_of_function_locals,
            )?
            .enforce_global_limit(self.max_number_of_globals)?
            .enforce_export_constraints(blueprints)?
            .inject_instruction_metering(&self.instrumenter_config)?
            .inject_stack_metering(self.instrumenter_config.max_stack_size())?
            .ensure_instantiatable()?
            .ensure_compilable()?
            .to_bytes()
    }
}
```

Once the module itself is statically validated, instrumented, and rewritten, the next step of validation is handed over to the specific VM implementation where the functions are instantiated, currently `WasmerModule::instantiate` and `WasmiModule::host_funcs_set`.

The linkage process is security sensitive as if the guest module is able to corrupt its own metadata (e.g., WASM spec is not followed and the guest code can export its own imports allowing the module to overwrite its own imports), it would enable DOS attacks. This is due to the fact that costing is handled via a callout to the `env.gas` function from the guest code, inserted at the time of instrumentation, which if corrupted, would lead to the execution-costing mechanism being bypassed.

Function: `WasmModule::enforce_import_constraints`

This is the method that validates the imports of the guest code,

```
pub fn enforce_import_constraints(&self, version: ScryptoVmVersion) ->
    Result<&Self, PrepareError>
```

which enforces the following constraints: 1) the imported function is valid for the given module and the version of the runtime it is linking against and 2) the type signature of the target function matches the imported function.

It achieves this by iterating over the imports of the module and switching on the import name:

```
// Only allow `env::radix_engine` import
for entry in self
    .module
```

```
        .import_section()
        .map_err(|err| PrepareError::ModuleInfoError(err.to_string()))?
        .unwrap_or(vec![])
    {
        if entry.module == MODULE_ENV_NAME {
            match entry.name {
                BUFFER_CONSUME_FUNCTION_NAME => {
                    if let TypeRef::Func(type_index) = entry.ty {
                        if Self::function_type_matches(
                            &self.module,
                            type_index,
                            vec![ValType::I32, ValType::I32],
                            vec![],
                        ) {
                            continue;
                        }

                        return Err(PrepareError::InvalidImport(

        InvalidImport::InvalidFunctionType(entry.name.to_string()),
                        ));
                    }
                }
                 ...
              }
        }
     }
```

Function: `WasmiModule::host_funcs_set`

This method finalizes the instantiation of the guest module by setting up the host functions that the guest module can call. It does so by instantiating each host function and then adding it to the linker. The WASMI linker then provides the appropriate bindings to invoke.

```
pub fn host_funcs_set(module: &Module, store: &mut Store<HostState>) ->
    Result<InstancePre, Error>
```

```
macro_rules! linker_define {
    ($linker: expr, $name: expr, $var: expr) => {
        $linker
            .define(MODULE_ENV_NAME, $name, $var)
            .expect(stringify!("Failed to define new linker item {}", $name));
    };
}
```

```
// ...

let host_consume_buffer = Func::wrap(
    store.as_context_mut(),
    |caller: Caller<'_, HostState>,
     buffer_id: BufferId,
     destination_ptr: u32|
     -> Result<(), Trap> {
        consume_buffer(caller, buffer_id,
    destination_ptr).map_err(|e| e.into())
    },
);

// ...

let mut linker = <Linker<HostState>>::new();
linker_define!(linker, BUFFER_CONSUME_FUNCTION_NAME, host_consume_buffer);
```

Switch statements

The switch statements in the `enforce_import_constraints` and `host_funcs_set` methods are a potential source of error as they are rather large and maintained by hand.

It is worth noting that a bug within these statements would most likely be caught by the comprehensive integration tests, but it remains a potential area for improvement as a proc macro could be used at the function-definition level to generate both the import validation and the runtime-specific bindings, eliminating the need for maintenance for both switch statements.

Costing

Costing in Radix combines both static analysis and runtime enforcement to ensure the fair and efficient use of computational and storage resources. Although we will mostly focus on execution-costing mechanisms in this section, subsystems roughly fall into the following categories:

1. Execution cost
    - Native-code execution
    - Guest-code execution (WASM)
2. Finalization cost
    - Commit events
    - Commit logs

- Commit state updates

Execution cost: Native code

Native-code execution here refers to the cost of executing any of the functions exported as the VM runtime. Unlike the guest code, the exact cost cannot be measured during the runtime for the sake of determinism, but the implemented mechanism comes very close to estimating the real cost of each call.

The costing mechanism is implemented as follows:

1. **Marking.** A proc macro is used at the specific functions that the guest code interacts with and is of interest to the costing runtime. It can also specify arguments to monomorphize the costing mechanism if any of them affect the cost of the call significantly (e.g., a function that processes a `Vec<T>` argument).
2. **Sampling.** When executed under QEMU, this macro times the execution of the function and writes a sample. After executing a number of sample transactions, the profiler writes a costing table to be used by production runtime.
3. **Costing.** Upon each call to the priorly marked functions, the costing runtime uses the costing table to estimate the cost of the call and charges it to the transaction.

While this is not an exact approach, it provides a reasonable approximation of the cost of each call.

Execution cost: Guest code (WASM)

The guest code's execution cost is measured and enforced by instrumenting the WASM module during the preparation phase. The instrumentation ensures that every operation in the WASM module is metered to track resource usage and enforce runtime limits.

1. **Validation phase**

   The `ScryptoV1WasmValidator` ensures that the WASM module adheres to predefined limits. It enforces constraints on the following:
   - Maximum memory size in pages
   - Maximum stack size
   - Maximum number of instructions
   - Maximum number of function parameters
   - Maximum number of function locals
   - Maximum number of tables
   - Maximum number of globals
   - Maximum number of functions
   - Maximum number of `br_table` targets

2. **Metering**

   The metering process involves several transformations to the WASM code to ensure proper resource tracking and safety:

   - The code is instrumented with additional instructions that call a `gas` function to deduct execution-cost units for each block of instructions.
   - Every function is wrapped into an indirect call stub to track the current stack height when used by `call_indirect` callers.
   - If the stack height exceeds the maximum limit, the execution halts to prevent stack overflows.

Given the following input WASM —

```
(module
  (type $sig (func (result i64)))
  (table 1 anyfunc)
  (elem (i32.const 0) $indirect)

  (func $indirect (type $sig)
    (i64.const 42)
  )

  (func $Test_f (param $0 i64) (result i64)
    (if (result i64)
      (i64.eqz (call_indirect (type $sig) (i32.const 0)))
      (then (i64.const 0))
      (else (i64.add (get_local $0) (i64.const 1)))
    )
  )

  (memory $0 1)
  (export "memory" (memory $0))
  (export "Test_f" (func $Test_f))
)
```

First, the control-flow graph is walked and the cost of each instruction within basic blocks are clustered, emitting a `gas` function prior.

```
;; instrumented version of $Test_f
(func (;2;) (type 1) (param i64) (result i64)
  i64.const 31251
  call 0 ;; the gas call for the first basic block (cost = 31251)
  i32.const 0
  call_indirect (type 0)
  i64.eqz
  if (result i64)
```

```
    i64.const 1372
    call 0 ;; the gas call for the first branch (cost = 1372)
    i64.const 0
  else
    i64.const 5811
    call 0 ;; the gas call for the second branch (cost = 5811)
    local.get 0
    i64.const 1
    i64.add
  end)
```

Afterwards, a thunk is created for each function, wrapping the instrumented function and adding the necessary stack checks.

```
;; thunk for $Test_f
(func (;4;) (type 1) (param i64) (result i64)
  local.get 0
  global.get 0 ;; stack height stored as a global variable
  i32.const 5 ;; 5 is the size of the stack frame for $Test_f
  i32.add
  global.set 0
  global.get 0
  i32.const 1024 ;; 1024 is the max stack height
  i32.gt_u
  if
    unreachable ;; trap if stack height is exceeded
  end
  call 2 ;; the actual $Test_f implementation
  global.get 0
  i32.const 5 ;; subtract the stack frame size and return
  i32.sub
  global.set 0)
```

From this point on, every reference to the `Test_f` function is replaced with the thunk, with the exception of direct calls where the stack guard is inlined into the flow.

Costing API

The runtime also exposes an API for querying costing parameters.

Function: `COSTING_GET_USD_PRICE`

This function retrieves the USD price for the use of stable costing. This is mainly used for fixed USD fees seen in blueprint royalties.

```
fn usd_price(&mut self) -> Result<u64, E>;
```

Function: `COSTING_GET_EXECUTION_COST_UNIT_LIMIT`

This function retrieves the execution-cost unit limit.

```
fn execution_cost_unit_limit(&mut self) -> Result<u32, E>;
```

Function: `COSTING_GET_EXECUTION_COST_UNIT_PRICE`

This function retrieves the execution-cost unit price.

```
fn execution_cost_unit_price(&mut self) -> Result<u64, E>;
```

Function: `COSTING_GET_FINALIZATION_COST_UNIT_LIMIT`

This function retrieves the finalization-cost unit limit.

```
fn finalization_cost_unit_limit(&mut self) -> Result<u32, E>;
```

Function: `COSTING_GET_FINALIZATION_COST_UNIT_PRICE`

This function retrieves the finalization-cost unit price.

```
fn finalization_cost_unit_price(&mut self) -> Result<u64, E>;
```

Function: `COSTING_GET_TIP_PERCENTAGE`

This function retrieves the tip percentage.

```
fn tip_percentage(&mut self) -> Result<u32, E>;
```

Function: `COSTING_GET_FEE_BALANCE`

This function retrieves the fee balance.

```
fn fee_balance(&mut self) -> Result<u64, E>;
```

Object operations

Creation and setup API

The Creation and Setup API provides fundamental operations for creating new objects and converting them into globally addressable entities. These operations form the foundation for object lifecycle management in the system.

`OBJECT_NEW`

This creates a new owned object instance within the caller's package.

```
fn new_object(
    &mut self,
    blueprint_ident: &str,
    features: Vec<&str>,
    generic_args: GenericArgs,
    fields: IndexMap<FieldIndex, FieldValue>,
    kv_entries: IndexMap<CollectionIndex, IndexMap<Vec<u8>, KVEntry>>,
) -> Result<NodeId, E>;
```

The operation includes the following validations:

1. **Package scoping**

   It ensures objects are only created within the caller's package scope:

   ```
   let package_address = actor
       .blueprint_id()
       .map(|b| b.package_address)

       .ok_or(RuntimeError::SystemError(SystemError::NoPackageAddress))?;
   ```

2. **Field validation**

   It checks the validity of all field indexes against the blueprint definition:

```
let expected_num_fields = blueprint_interface.state.num_fields();
for field_index in fields.keys() {
    let field_index: usize = (*field_index) as u32 as usize;
    if field_index >= expected_num_fields {
        return
    Err(RuntimeError::SystemError(SystemError::CreateObjectError(...)));
    }
}
```

Function: `OBJECT_GLOBALIZE`

This converts an owned object into a globally addressable one, attaching required modules and validating access.

```
fn globalize(
    &mut self,
    node_id: NodeId,
    modules: IndexMap<AttachedModuleId, NodeId>,
    address_reservation: Option<GlobalAddressReservation>
) -> Result<GlobalAddress, E>
```

The operation performs the following validations and checks:

1. **Package authorization**

   It ensures only the original package can globalize its objects:

   ```
   if Some(reserved_blueprint_id.package_address)
       != actor.package_address() {
      return Err(RuntimeError::SystemError(

       SystemError::InvalidGlobalizeAccess(Box::new(InvalidGlobalizeAccess
       {
               package_address: reserved_blueprint_id.package_address,
               blueprint_name: reserved_blueprint_id.blueprint_name,
               actor_package: actor.package_address(),
         })),
      ));
   }
   ```

2. **Required modules**

   It confirms mandatory modules (`RoleAssignment` and `Metadata`) are included:

```
if !modules.contains_key(&AttachedModuleId::RoleAssignment) {
   return Err(RuntimeError::SystemError(SystemError::MissingModule(
      ModuleId::RoleAssignment,
   )));
}
if !modules.contains_key(&AttachedModuleId::Metadata) {
   return Err(RuntimeError::SystemError(SystemError::MissingModule(
      ModuleId::Metadata,
   )));
}
```

3. **Address reservation validation**

   It validates the global address reservation:

```
match type_info {
   Some(TypeInfoSubstate::GlobalAddressReservation(x)) => x,
   _ => {
      return Err(RuntimeError::SystemError(
            SystemError::InvalidGlobalAddressReservation,
      ));
   }
}
```

4. **Module-type validation**

   It ensures attached modules align with their expected types:

```
let blueprint_id =
    self.get_object_info(node_id)?.blueprint_info.blueprint_id;
let expected_blueprint = module_id.static_blueprint();
if !blueprint_id.eq(&expected_blueprint) {
   return
    Err(RuntimeError::SystemError(SystemError::InvalidModuleType(
      Box::new(InvalidModuleType {
            expected_blueprint,
            actual_blueprint:  blueprint_id,
      }),
   )));
}
```

5. **Transient blueprint check**

   It prevents globalization of transient blueprints:

```
if blueprint_definition.interface.is_transient {
   return Err(RuntimeError::SystemError(
      SystemError::GlobalizingTransientBlueprint,
    ));
}
```

Invocation API

The Invocation API enables invocation of functions on foreign blueprints or modules.

Function: `OBJECT_CALL`

This method enables the standard method invocation path on objects in the system — `SystemModuleMixer::on_call_method()`, and hence `AuthModule`, is used to establish an auth zone and resolve permissions, and the call itself is delegated to the `kernel_invoke()` function after.

```
fn call_method(
    &mut self,
    receiver: &NodeId,
    method_name: &str,
    args: Vec<u8>,
) -> Result<Vec<u8>, E>;
```

1. **Authorization setup**

   It creates an auth zone for the current context using `create_auth_zone()`, retrieves the blueprint info of the `receiver`, and resolves the method permissions via `resolve_method_permission()`.

2. **Permission checks**

   Resolved permissions are checked through `check_permission()`, which validates access rules or role lists, enforces role-authorization boundaries, and returns an `Unauthorized` error on failure.

3. **Function invocation**

   It passes the validated function call into the engine for execution using `kernel_invoke()`.

4. **Cleanup**

   It performs the required teardown via `teardown_auth_zone()`, detaching proofs from the auth zone, dropping owned proofs, and destroying the auth zone itself.

5. **Execution result**

   It returns SBOR-encoded data or an `InvokeError` for failures.

Function: `BLUEPRINT_CALL`

This method facilitates blueprint-level invocations, independent of any specific object instance. Unlike `OBJECT_CALL`, it does not rely on object states.

```
fn call_function(
    &mut self,
    package_address: PackageAddress,
    blueprint_name: &str,
    function_name: &str,
    args: Vec<u8>,
) -> Result<Vec<u8>, E>;
```

1. **Preparation and validation**

   It constructs a `BlueprintId` from the `package_address` and `blueprint_name`. Auth-zone creation is more generalized and not tied to a specific object instance.

2. **Blueprint permission resolution**

   It uses `SystemModuleMixer::on_call_function` to validate the package and blueprint existence. It also checks function accessibility rules defined at the blueprint level. Functions have no internal role enforcement but can include custom access checks using proofs or external authorization mechanisms.

3. **Function invocation**

   It passes the validated function call into the engine for execution using `kernel_invoke()`.

4. **Cleanup**

   It performs the required teardown via `teardown_auth_zone()`, similar to `OBJECT_CALL`.

5. **Execution result**

   It returns SBOR-encoded data or an `InvokeError` for failures.

Function: `OBJECT_CALL_MODULE`

```
fn call_module_method(
    &mut self,
    receiver: &NodeId,
    module_id: AttachedModuleId,
    method_name: &str,
    args: Vec<u8>,
) -> Result<Vec<u8>, E>;
```

This method enables calls to attached modules (`Metadata`, `Royalty`, `RoleAssignment`) on global objects. Each module enforces its own authorization templates and rules.

1. **Module validation**

   It verifies that the object has **global** status, checks for module existence in the object's module map, ensures the module type aligns with system constraints, and rejects early if module requirements are not satisfied.

2. **Authorization setup**

   It creates an auth zone specific to the module context, maps the `module_id` to the corresponding blueprint info, resolves the correct auth template for the module type, and handles additional requirements (e.g., `RoleAssignment` requiring global address context).

3. **Actor configuration**

   It configures the actor for the module-specific call.

   ```
   Actor::Method(MethodActor {
       method_type: MethodType::Module(module_id), // Key distinction
       node_id: receiver.clone(),
       ident: method_name.to_string(),
       auth_zone: auth_actor_info.clone(),
       object_info,
   })
   ```

4. **Partition isolation**

   Module partitions are isolated from each other, ensuring the key space remains collision-free within the context of the same object boundary as well as preventing cross-module state access.

Function: `OBJECT_CALL_DIRECT`

This method is used for invocations similar to `object_call()` but provides direct access to objects, bypassing certain authorization checks. It is specifically designed for high-privilege operations on objects like vaults.

```
fn call_direct_access_method(
    &mut self,
    receiver: &NodeId,
    method_name: &str,
    args: Vec<u8>,
) -> Result<Vec<u8>, E>;
```

1. **Receiver-type validation**

It validates that the receiver explicitly allows direct access.

```
fn is_direct_access_receiver(receiver: &Option<ReceiverInfo>) -> bool {
    if let Some(ref receiver) = receiver {
        match (&receiver.receiver, receiver.ref_types) {
            (Receiver::SelfRef | Receiver::SelfRefMut,
    RefTypes::DIRECT_ACCESS) => true,
            _ => false,
        }
    } else {
        false
    }
}
```

2. **Blueprint access control**

Only specific system blueprints (like vaults) allow direct access.

```
if blueprint_id.package_address.eq(&RESOURCE_PACKAGE)
    && (blueprint_id.blueprint_name.eq(FUNGIBLE_VAULT_BLUEPRINT)
        || blueprint_id.blueprint_name.eq(NON_FUNGIBLE_VAULT_BLUEPRINT))
{
    Ok(StableReferenceType::DirectAccess)
}
```

3. **Method actor setup**

It creates a special method actor with the `Direct` method type.

```
Actor::Method(MethodActor {
    method_type: MethodType::Direct,  // Key distinction
    node_id: receiver.clone(),
    ident: method_name.to_string(),
    auth_zone: auth_actor_info.clone(),
    object_info,
})
```

4. **Authorization resolution**

Direct access methods have specific restrictions around global addressing.

```
match module_id {
    ModuleId::Main => {
        if is_direct_access {
            return Err(RuntimeError::SystemError(
               SystemError::GlobalAddressDoesNotExist,
            ));
        }
```

Address operations

The address API provides utilities for global address management.

Function: `ADDRESS_ALLOCATE`

This handles global address allocation.

```
fn allocate_address(&mut self) -> Result<GlobalAddressReservation, E>;
```

1. **Uniqueness guarantee**

   It ensures globally unique address generation.

   ```
   let address = self.address_allocator.next_address()?;
   if self.address_exists(address) {
       return
       Err(RuntimeError::SystemError(SystemError::AddressCollision));
   }
   ```

2. **Reservation management**

   It tracks address reservations until used.

   ```
   self.reserved_addresses.insert(
       address,
       ReservationInfo {
           reserved_at: self.current_epoch(),
           reserved_by: self.get_current_actor()
       }
   );
   ```

Function: `ADDRESS_GET_RESERVATION_ADDRESS`

This retrieves an address from the reservation:

```
fn get_reservation_address(&self, reservation: &GlobalAddressReservation) ->
    Result<GlobalAddress, E>;
```

1. **Reservation validation**

   It verifies a reservation exists and has not expired.

   ```
   if !self.is_valid_reservation(reservation) {
       return
       Err(RuntimeError::SystemError(SystemError::InvalidReservation));
   }
   ```

2. **Address resolution**

   It maps a reservation to a concrete address.

   ```
   match self.reserved_addresses.get(&reservation.id) {
       Some(info) => Ok(info.address),
       None =>
       Err(RuntimeError::SystemError(SystemError::ReservationNotFound))
   }
   ```

3. **Life cycle management**

   It tracks reservation usage and expiration.

   ```
   if self.is_reservation_expired(reservation) {
       self.reserved_addresses.remove(&reservation.id);
       return
       Err(RuntimeError::SystemError(SystemError::ReservationExpired));
   }
   ```

Utility API

The Utility API provides a set of helper methods for querying and validating object properties, relationships, and type information. These methods support common object inspection and verification tasks.

Function: `OBJECT_INSTANCE_OF`

This method checks if an object is an instance of a specific blueprint by comparing the object's blueprint ID with the provided package address and blueprint name.

```
fn instance_of(&mut self, object_id: Vec<u8>, package_address: Vec<u8>,
    blueprint_name: Vec<u8>) -> Result<u32, E>
```

Function: `OBJECT_GET_BLUEPRINT_ID`

This method retrieves the blueprint ID of an object though the `get_object_info()` method.

```
fn get_blueprint_id(&mut self, node_id: &NodeId) -> Result<BlueprintId, E>;
```

Function: `OBJECT_GET_OUTER_OBJECT`

This method retrieves the outer object of an object through the `get_object_info()` method.

```
fn get_outer_object(&mut self, node_id: &NodeId) -> Result<GlobalAddress, E>;
```

Key-value store operations

The key-value store API implements comprehensive validation and access control through

- type safety through schema validation,
- access control via locking mechanisms, and
- resource cleanup through handle management.

Constructor API

The Constructor API provides methods for creating and initializing new key-value stores with proper schema validation and type safety controls.

`KEY_VALUE_STORE_NEW`

This creates a new key-value store with schema validation.

```
fn key_value_store_new(
    &mut self,
    data_schema: KeyValueStoreDataSchema,
) -> Result<NodeId, RuntimeError>;
```

The following security controls are in place:

1. **Schema validation**

   It validates the provided schema with built-in validation rules before store creation and prevents invalid schemas from being used.

```
validate_schema(additional_schema.v1())
    .map_err(|_| RuntimeError::SystemError(SystemError::InvalidGenericArgs))?;
```

2. **Type safety**

   It validates generic type substitutions against schema and ensures type safety across all store operations.

```
self.validate_kv_store_generic_args(&additional_schemas, &key_type,
    &value_type)
    .map_err(|e| RuntimeError::SystemError(SystemError::TypeCheckError(e)))?;
```

Handle API

The Handle API manages the lifecycle of key-value store entries through controlled access mechanisms, including opening entries with appropriate locks and proper cleanup of resources.

`KEY_VALUE_STORE_OPEN_ENTRY`

This opens a key-value entry with appropriate locking.

```
fn key_value_store_open_entry(
    &mut self,
    node_id: &NodeId,
    key: &Vec<u8>,
    flags: LockFlags,
) -> Result<KeyValueEntryHandle, RuntimeError>;
```

The key safeguards include the following:

1. **Type verification**

   It verifies the type of the node before accessing it to prevent type-confusion attacks.

```
match type_info {
    TypeInfoSubstate::KeyValueStore(info) => info,
    _ =>
    return Err(RuntimeError::SystemError(SystemError::NotAKeyValueStore)),
}
```

2. **Lock-flag validation**

   It validates lock flags against allowed operations. This prevents internal lock flags from being used in the user-facing API, which could lead to unintended behavior.

```
if flags.contains(LockFlags::UNMODIFIED_BASE)
    || flags.contains(LockFlags::FORCE_WRITE) {
    return Err(RuntimeError::SystemError(SystemError::InvalidLockFlags));
}
```

3. **Key schema validation**

   It validates the key against the store's schema before access.

```
self.validate_kv_store_payload(&target, KeyOrValue::Key, &key)?;
```

4. **Lock-status checks**

   It checks the lock status before granting write access, preventing write access to locked entries. This enables implementation of write-once semantics.

```
if flags.contains(LockFlags::MUTABLE) {
    let lock_status = self.api.kernel_read_substate(handle).map(|v| {
        let kv_entry: KeyValueEntrySubstate<ScryptoValue>
    = v.as_typed().unwrap();
        kv_entry.lock_status()
    })?;

    if let LockStatus::Locked = lock_status {
        return
    Err(RuntimeError::SystemError(SystemError::KeyValueEntryLocked));
    }
}
```

`KEY_VALUE_ENTRY_CLOSE`

This manages entry-handle cleanup.

```
fn key_value_entry_close(&mut self, handle: KeyValueEntryHandle) -> Result<(),
    RuntimeError>;
```

Entry API

The Entry API allows the reading and writing of key-value store entries.

`KEY_VALUE_ENTRY_READ` / `KEY_VALUE_ENTRY_WRITE`

These control read/write access to entries.

```
fn key_value_entry_read(&mut self, handle: KeyValueEntryHandle) ->
    Result<Vec<u8>, RuntimeError>;
fn key_value_entry_write(&mut self, handle: KeyValueEntryHandle, buffer:
    Vec<u8>) -> Result<(), RuntimeError>;
```

The following are the key protections:

1. **Lock validation**
   It verifies the appropriate lock type for an operation and prevents handle type-confusion attacks.

```
match data {
    SystemLockData::KeyValueEntry(..) => {}
    _ => return
    Err(RuntimeError::SystemError(SystemError::NotAKeyValueEntryHandle));
}
```

2. **Write-permission check**
   It validates write permissions before modification and enforces proper access control.

```
match data {
    SystemLockData::KeyValueEntry(KeyValueEntryLockData::KVStoreWrite { .. })
    => {}
    _ => return
    Err(RuntimeError::SystemError(SystemError::NotAKeyValueEntryWriteHandle));
}
```

3. **Schema validation on write**

   It validates written data against a store schema and prevents invalid data corruption from entering the store.

```
self.validate_kv_store_payload(
    &kv_store_validation_target,
    KeyOrValue::Value,
    &buffer,
)?;
```

KEY_VALUE_ENTRY_REMOVE / KEY_VALUE_STORE_REMOVE_ENTRY

The first method removes an already opened entry, whereas the latter removes an entry from the store by a `(node, key)` pair reference.

```
fn key_value_entry_remove(&mut self, handle: KeyValueEntryHandle) ->
    Result<Vec<u8>, RuntimeError>;
fn key_value_store_remove_entry(&mut self, node_id: &NodeId, key: &Vec<u8>) ->
    Result<Vec<u8>, RuntimeError>;
```

There is one main security control:

1. **Write-lock requirement**

   It requires a write lock for removal and prevents immutable entries from being deleted.

```
if !data.is_kv_entry_with_write() {
    return
    Err(RuntimeError::SystemError(SystemError::NotAKeyValueEntryWriteHandle));
}
```

Actor operations

Actor field API

The Actor Field API provides operations for managing access to actor object fields, enabling controlled reading and writing of field data while maintaining state consistency and access control.

`ACTOR_OPEN_FIELD`

This opens a field on the current actor object for reading or writing.

```
fn open_field(
    &mut self,
    field_index: FieldIndex,
) -> Result<SubstateHandle, E>;
```

It includes critical validations:

1. **Actor-context validation**

   It ensures operation occurs within valid actor context.

   ```
   if self.get_actor().actor_type() != ActorType::Method {
       return
       Err(RuntimeError::SystemError(SystemError::InvalidActorContext));
   }
   ```

2. **Field-index validation**

   It validates the field index against the blueprint definition.

   ```
   let num_fields = self.get_blueprint_interface().state.num_fields();
   if field_index >= num_fields {
       return
       Err(RuntimeError::SystemError(SystemError::InvalidFieldIndex));
   }
   ```

3. **Duplicate open prevention**

   It prevents opening already opened fields.

```
if self.is_field_opened(field_index) {
    return Err(RuntimeError::SystemError(SystemError::FieldAlreadyOpened));
}
```

FIELD_ENTRY_READ / FIELD_ENTRY_WRITE / FIELD_ENTRY_CLOSE

These operations manage field access after opening:

```
fn read_field(&mut self, handle: SubstateHandle) -> Result<Vec<u8>, E>;
fn write_field(&mut self, handle: SubstateHandle, data: Vec<u8>) -> Result<(),
    E>;
fn close_field(&mut self, handle: SubstateHandle) -> Result<(), E>;
```

The field operations perform the following validations and state management:

1. **Handle validation**

   It verifies a handle corresponds to an opened field.

```
if !self.is_valid_handle(handle) {
    return Err(RuntimeError::SystemError(SystemError::InvalidHandle));
}
```

2. **Access control**

   Read requires a field to be opened, write requires mutable access, and close releases a field lock.

```
match self.get_field_access_type(handle) {
    AccessType::ReadOnly if is_write => {
        return Err(RuntimeError::SystemError(SystemError::ReadOnlyField));
    }
    _ => {}
}
```

3. **State consistency**

   It maintains field open/close state tracking.

```
self.opened_fields.remove(&handle);
if self.opened_fields.is_empty() {
    self.clear_actor_state();
}
```

Actor information API

The Actor Information API provides methods to retrieve identity and contextual information about the currently executing actor, including its object ID, package address, and blueprint name.

`ACTOR_GET_OBJECT_ID` / `ACTOR_GET_PACKAGE_ADDRESS` / `ACTOR_GET_BLUEPRINT_NAME`

These methods provide identity information about the current actor:

```
fn get_object_id(&self) -> Result<ObjectId, E>;
fn get_package_address(&self) -> Result<PackageAddress, E>;
fn get_blueprint_name(&self) -> Result<String, E>;
```

These methods perform the following validations and access control checks:

1. **Context validation**

   It ensures operations occur in valid actor context.

```
   if !self.has_active_actor() {
       return Err(RuntimeError::SystemError(SystemError::NoActiveActor));
   }
  ```

2. **Information access control**
    Object ID access is restricted to method context,
    and package/blueprint info is available in function context.
```rs
   match self.get_actor_type() {
       ActorType::Method => Ok(self.get_current_object_id()),
       _ => Err(RuntimeError::SystemError(SystemError::InvalidActorType))
   }
```

Actor event API

The Actor Event API enables actors to emit events during execution, providing a mechanism for tracking and logging important state changes or notifications with appropriate validation and size constraints.

`ACTOR_EMIT_EVENT`

This handles event emission from actors:

```
fn emit_event(&mut self, event_name: &str, event_data: Vec<u8>) -> Result<(),
    E>;
```

The event emission process includes the following validations and tracking:

1. **Event-schema validation**

   It validates the event name and data format.

```
if !is_valid_event_name(event_name) {
    return Err(RuntimeError::SystemError(SystemError::InvalidEventName));
}
```

2. **Size limits**

   It enforces size restrictions on event data.

```
if event_data.len() > MAX_EVENT_SIZE {
    return Err(RuntimeError::SystemError(SystemError::EventSizeExceeded));
}
```

3. **Context tracking**

   It records the event in the system's event log with the current actor context.

```
self.event_log.push(Event {
    emitter: self.get_current_actor(),
    name: event_name.to_string(),
    data: event_data
});
```

Utility functions

Crypto API

Crypto API provides a set of cryptographic functions for use within blueprints — `blst` is used internally to implement BLS12-381 elliptic-curve operations.

Function: `CRYPTO_UTILS_BLS12381_V1_VERIFY`

This function verifies a BLS12-381 signature for a single message.

```
fn bls12381_v1_verify(
    &mut self,
    message: &[u8],
    public_key: &Bls12381G1PublicKey,
    signature: &Bls12381G2Signature,
) -> Result<u32, E>;
```

Function: `CRYPTO_UTILS_BLS12381_V1_AGGREGATE_VERIFY`

This function verifies an aggregate BLS12-381 signature for multiple messages.

```
fn bls12381_v1_aggregate_verify(
    &mut self,
    pub_keys_and_msgs: &[(Bls12381G1PublicKey, Vec<u8>)],
    signature: &Bls12381G2Signature,
) -> Result<u32, E>;
```

Function: `CRYPTO_UTILS_BLS12381_V1_FAST_AGGREGATE_VERIFY`

This function verifies a BLS12-381 signature for a single message with multiple public keys.

```
fn bls12381_v1_fast_aggregate_verify(
    &mut self,
    message: &[u8],
    public_keys: &[Bls12381G1PublicKey],
    signature: &Bls12381G2Signature,
) -> Result<u32, E>;
```

Function: `CRYPTO_UTILS_BLS12381_G2_SIGNATURE_AGGREGATE`

This function aggregates multiple BLS12-381 signatures into one by adding them together.

```
fn bls12381_g2_signature_aggregate(
    &mut self,
    signatures: &[Bls12381G2Signature],
) -> Result<Bls12381G2Signature, E>;
```

Function: `CRYPTO_UTILS_KECCAK256_HASH`

This function returns the Keccak-256 hash of the input data.

```
fn keccak256_hash(&mut self, data: &[u8]) -> Result<Hash, E>;
```

Memory-management API

Function: `BUFFER_CONSUME`

This function consumes a buffer and copies its contents to the specified destination pointer. The size is not necessary to communicate at this point as the caller holds this information prior to calling this function, as any API returning a buffer also returns the size.

Buffers are used to pass data between the host and the guest code. The system tracks any returned buffers through an index map local to the current VM instance. This approach has the potential to create a memory-exhaustion issue due to the buffer limits being renewed on each stack frame.

```
fn consume_buffer(&mut self, buffer_id: BufferId, destination_ptr: u32) ->
    Result<(), E>;
```

System API

Function: `SYS_LOG`

This function emits a log message with a specified severity level. The message size is constrained by the costing model, so it presents no risks.

```
fn emit_log(&mut self, level: Level, message: String) -> Result<(), E>;
```

Function: `SYS_BECH32_ENCODE_ADDRESS`

This function encodes a global address into Bech32 format.

```
fn bech32_encode_address(&mut self, address: GlobalAddress) -> Result<String,
    E>;
```

Function: `SYS_PANIC`

This function triggers a system panic with an error message.

```
fn panic(&mut self, message: String) -> Result<(), E>;
```

Function: `SYS_GET_TRANSACTION_HASH`

This function retrieves the hash of the current transaction.

```
fn get_transaction_hash(&mut self) -> Result<Hash, E>;
```

Function: `SYS_GENERATE_RUID`

This function generates a Radix unique identifier (RUID). This is done by combining parts of the transaction hash as well as a block local ID counter.

```
fn generate_ruid(&mut self) -> Result<[u8; 32], E>;
```

5.3. Module: Transactions

Transaction processor

When a transaction is being executed by the Radix engine, the initial entry point is a native blueprint called the transaction processor. It acts as a dispatcher for the transaction manifest and invokes the appropriate blueprints to execute the user instructions.

```
fn start<Y>(...) -> Result<Vec<InstructionOutput>, RuntimeError> {
    // ...Prepare system, allocate global addresses

    let rtn = system.call_function(
        TRANSACTION_PROCESSOR_PACKAGE,
        TRANSACTION_PROCESSOR_BLUEPRINT,
        TRANSACTION_PROCESSOR_RUN_IDENT,
        scrypto_encode(&TransactionProcessorRunInputEfficientEncodable {
            manifest_encoded_instructions,
            global_address_reservations,
            references,
            blobs,
        })
```

```
            .unwrap(),
    )?;

    // ...
}
```

Special privileges

The transaction-processor blueprint is more privileged than other blueprints with certain restrictions on user resources relaxed by exceptions being made for it, both at the engine level as well as at the access-control level of other native blueprints.

Some exceptions made for the transaction processor include the following:

1. **Publishing a new package**

   This operation is restricted only to the transaction processor by the following access rule:

```
FunctionAuth::AccessRules(
    indexmap!(
        PACKAGE_PUBLISH_WASM_IDENT.to_string() =>
    rule!(require(package_of_direct_caller(TRANSACTION_PROCESSOR_PACKAGE))),
        PACKAGE_PUBLISH_WASM_ADVANCED_IDENT.to_string() =>
    rule!(require(package_of_direct_caller(TRANSACTION_PROCESSOR_PACKAGE))),
        PACKAGE_PUBLISH_NATIVE_IDENT.to_string() =>
    rule!(require(AuthAddresses::system_role())),
    )
),
```

2. **Auth-zone creation**

   Auth zones created by a call from the transaction processor start with the initial proofs and virtual resources as opposed to being empty.

```
let auth_zone = {
   // TODO: Remove special casing use of transaction processor and just have
    virtual resources
   // stored in root call frame
   let is_transaction_processor_blueprint = blueprint_id
       .package_address
       .eq(&TRANSACTION_PROCESSOR_PACKAGE)
       && blueprint_id
           .blueprint_name
```

```
            .eq(TRANSACTION_PROCESSOR_BLUEPRINT);
    let is_at_root = api.kernel_get_current_depth() == 0;
    let (virtual_resources, virtual_non_fungibles) =
        if is_transaction_processor_blueprint && is_at_root {
            let auth_module = &api.kernel_get_system().modules.auth;
            (
                auth_module.params.virtual_resources.clone(),
                auth_module.params.initial_proofs.clone(),
            )
        } else {
            (BTreeSet::new(), BTreeSet::new())
        };
    Self::create_auth_zone(api, None, virtual_resources,
     virtual_non_fungibles)?
};
```

Dispatcher loop

This loop forms the core of transaction execution, translating high-level manifest instructions into actual state changes in the ledger through the Radix Engine.

```
fn TransactionProcessorBlueprint::run<Y, L: Default>(
  manifest_encoded_instructions: Vec<u8>,
  global_address_reservations: Vec<GlobalAddressReservation>,
  _references: Vec<Reference>,
  blobs: IndexMap<Hash, Vec<u8>>,
  version: TransactionProcessorV1MinorVersion,
  api: &mut Y,
) -> Result<Vec<InstructionOutput>, RuntimeError>
```

The dispatcher loop consists of three main phases that handle the complete lifecycle of transaction processing:

1. **Initialization**

   It creates a worktop component, decodes the manifest instructions from the input bytes, and initializes a `TransactionProcessor` instance with provided blobs and address reservations.

2. **Main processing loop**

   It iterates through each instruction in the manifest sequentially, tracks current instruction index for execution tracing and error handling, processes each instruction based on its variant type through a large match statement, and collects outputs from each instruction execution for the receipt.

3. **Output handling**
   Each instruction execution may produce an `InstructionOutput`. Outputs are collected in order of execution. Any failures during processing will terminate execution and return an error.

Transaction manifests

A transaction manifest is a sequence of instructions that defines operations to be executed on the Radix Engine. It serves as a programmatic description of transaction intent, allowing users and developers to interact with the network in a structured way.

These are the key manifest components:

- Instructions for resource manipulation (buckets, proofs)
- Function and method calls to components/blueprints
- Authorization and access-control operations
- Network-component interactions
- Address-management operations

The manifest processor ensures secure execution through the following:

- Requiring explicit intent of the user to manage resources
- Resource tracking and isolation
- Authorization verification
- State-consistency checks
- Size and complexity limits

System-wide security considerations

The primary security consideration at this stage is ensuring that the stack frames are appropriately created so that the privileges granted to the transaction processor are not inadvertently leaked to other blueprints. This is trivially achieved due to the explicit nature of the calls, but it is important to take into consideration for any future changes to the system that may introduce new callout mechanisms.

Other considerations include the following:

- Proofs must be validated before use.
- Operations must verify appropriate authorization.
- Access-control rules must be enforced manually if acting on behalf of another component.
- Auto moves (`handle_call_return_data`) into worktop must be handled carefully (i.e., nodes owned by `(NON_)FUNGIBLE_BUCKET_BLUEPRINT`).

Preview execution

Preview execution allows testing the manifest without committing to the network. This is useful for development but also for users to understand the effects of their transactions before committing through the help of their wallet software.

The manifest system supports preview execution through flags:

```
struct PreviewFlags {
    use_free_credit: bool,
    assume_all_signature_proofs: bool,
    skip_epoch_check: bool,
    disable_auth: bool
}
```

Aliased operations

Some operations that are available in the manifest builder or the AST format do not map to a unique instruction. Instead, they are aliases for a `call_function` / `call_method` / `call_direct_vault_method` operation with a specific set of arguments.

Aliases bring convenience and readability to the manifest and consistency with less boilerplate to the transaction processor, but they may hide the underlying operation from a casual reader. Understanding both the aliases and their underlying instructions can be helpful for debugging and advanced usage.

Resource-management aliases

Alias	Translates to
`MintFungible`	`CallFunction { ResourceAddress, "Fungible", "mint" }`
`MintNonFungible`	`CallFunction { ResourceAddress, "NonFungible", "mint" }`
`MintRuidNonFungible`	`CallFunction { ResourceAddress, "NonFungible", "mint_ruid" }`

Here is an example:

```
// Alias
```

```
MINT_FUNGIBLE Address("resource_sim1...") Decimal("100");

// Translates to
CALL_METHOD Address("resource_sim1...") "mint" Decimal("100");
```

Vault operations aliases

Alias	Translates to
`RecallFromVault`	`CallDirectVaultMethod { "recall" }`
`FreezeVault`	`CallDirectVaultMethod { "freeze" }`
`UnfreezeVault`	`CallDirectVaultMethod { "unfreeze" }`
`RecallNonFungiblesFromVault`	`CallDirectVaultMethod { "recall_non_fungibles" }`

Here is an example:

```
// Alias
RECALL_FROM_VAULT Address("internal_sim1...") Decimal("50");

// Translates to
CALL_DIRECT_VAULT_METHOD Address("internal_sim1...") "recall" Decimal("50");
```

Package and component aliases

Alias	Translates to
`CreateValidator`	`CallFunction { ConsensusManager, "Validator", "create" }`
`PublishPackage`	`CallFunction { PackagePackage, "Package", "publish" }`
`PublishPackageAdvanced`	`CallFunction { PackagePackage, "Package", "publish_advanced" }`
`ClaimPackageRoyalties`	`CallMethod { method_name: "claim_royalties" }`

Here is an example:

```
// Alias
PUBLISH_PACKAGE Blob("a710f0...") Map<String, Tuple>();

// Translates to
CALL_FUNCTION Address("package_sim1...") "Package" "publish" Blob("a710f0...")
    Map<String, Tuple>();
```

Resource creation aliases

Alias	Translates to
`CreateFungibleResource`	`CallFunction { ResourcePackage, "Fungible", "create" }`
`CreateFungibleResourceWithInitialSupply`	`CallFunction { ResourcePackage, "Fungible", "create_with_initial_supply" }`
`CreateNonFungibleResource`	`CallFunction { ResourcePackage, "NonFungible", "create" }`
`CreateNonFungibleResourceWithInitialSupply`	`CallFunction { ResourcePackage, "NonFungible", "create_with_initial_supply" }`

Here is an example:

```
// Alias
CREATE_FUNGIBLE_RESOURCE Enum<0u8>() false 18u8 Tuple(...);

// Translates to
CALL_FUNCTION Address("resource_sim1...") "Fungible" "create" Enum<0u8>()
    false 18u8 Tuple(...);
```

Core instructions

1. Worktop operations

These instructions manage resources on the transaction worktop — a temporary storage area that tracks resources during transaction execution.

Instruction: `TakeFromWorktop`

Signature: `TakeFromWorktop { resource_address: ResourceAddress, amount: Decimal }`

This takes a specific amount of fungible resources from the worktop and creates a new bucket.

Instruction: `TakeAllFromWorktop`

Signature: `TakeAllFromWorktop { resource_address: ResourceAddress }`

This takes all available resources of a specified type from the worktop.

Instruction: `TakeNonFungiblesFromWorktop`

Signature: `TakeNonFungiblesFromWorktop { resource_address: ResourceAddress, ids: Vec<NonFungibleLocalId> }`

This takes specific NFTs from the worktop by ID.

Instruction: `ReturnToWorktop`

Signature: `ReturnToWorktop { bucket_id: ManifestBucket }`

This returns resources from a bucket to the worktop.

Instruction: `AssertWorktopContains`

Signature: `AssertWorktopContains { resource_address: ResourceAddress, amount: Decimal }`

This verifies the worktop contains at least a specified amount of resources.

Instruction: `AssertWorktopContainsAny`

Signature: `AssertWorktopContainsAny { resource_address: ResourceAddress }`

This verifies the worktop contains any amount of a specified resource.

Instruction: `AssertWorktopContainsNonFungibles`

Signature: `AssertWorktopContainsNonFungibles { resource_address: ResourceAddress, ids: Vec<NonFungibleLocalId> }`

This verifies the worktop contains specified NFT IDs.

2. Authorization operations

These instructions manage proofs and access control through the authorization zone.

Instruction: `PopFromAuthZone`

Signature: `PopFromAuthZone`

This removes and returns the most recently added proof from the authorization zone.

Instruction: `PushToAuthZone`

Signature: `PushToAuthZone { proof_id: ManifestProof }`

This adds a proof to the auth zone.

Instruction: `CreateProofFromAuthZoneOfAmount`

Signature: `CreateProofFromAuthZoneOfAmount { resource_address: ResourceAddress, amount: Decimal }`

This creates a proof for a specified fungible amount.

Instruction: `CreateProofFromAuthZoneOfNonFungibles`

Signature: `CreateProofFromAuthZoneOfNonFungibles { resource_address: ResourceAddress, ids: Vec<NonFungibleLocalId> }`

This creates a proof for specified NFTs.

Instruction: `CreateProofFromAuthZoneOfAll`

Signature: `CreateProofFromAuthZoneOfAll { resource_address: ResourceAddress }`

This creates a proof for all resources of a type.

Instruction: `DropAuthZoneProofs`

Signature: `DropAuthZoneProofs`

This removes all proofs from the auth zone.

Instruction: `DropAuthZoneRegularProofs`

Signature: `DropAuthZoneRegularProofs`

This removes non-signature proofs.

Instruction: `DropAuthZoneSignatureProofs`

Signature: `DropAuthZoneSignatureProofs`

This removes signature proofs.

3. Bucket operations

These instructions manipulate resource buckets directly.

Instruction: `CreateProofFromBucketOfAmount`

Signature: `CreateProofFromBucketOfAmount { bucket_id: ManifestBucket, amount: Decimal }`

This creates a proof for specified amount from a bucket.

Instruction: `CreateProofFromBucketOfNonFungibles`

Signature: `CreateProofFromBucketOfNonFungibles { bucket_id: ManifestBucket, ids: Vec<NonFungibleLocalId> }`

This creates a proof for specified NFTs from a bucket.

Instruction: `CreateProofFromBucketOfAll`

Signature: `CreateProofFromBucketOfAll { bucket_id: ManifestBucket }`

This creates a proof for the entire bucket contents.

Instruction: `BurnResource`

Signature: `BurnResource { bucket_id: ManifestBucket }`

This burns resources in a bucket.

4. Proof operations

These instructions manage proof lifecycle including creation, copying, and destruction of proofs.

Instruction: `CloneProof`

Signature: `CloneProof { proof_id: ManifestProof }`

This creates a copy of an existing proof.

Instruction: `DropProof`

Signature: `DropProof { proof_id: ManifestProof }`

This destroys a specified proof.

5. Method/function calls

Instruction: `CallFunction`

Signature: `CallFunction { package_address: DynamicPackageAddress, blueprint_name: String, function_name: String, args: ManifestValue }`

This calls the blueprint function.

Instruction: `CallMethod`

Signature: `CallMethod { address: DynamicGlobalAddress, method_name: String, args: ManifestValue }`

This calls a component method.

Instruction: `CallRoyaltyMethod`

Signature: `CallRoyaltyMethod { address: DynamicGlobalAddress, method_name: String, args: ManifestValue }`

This calls a royalty-module method.

Instruction: `CallMetadataMethod`

Signature: `CallMetadataMethod { address: DynamicGlobalAddress, method_name: String, args: ManifestValue }`

This calls a metadata module method.

Instruction: `CallRoleAssignmentMethod`

Signature: `CallRoleAssignmentMethod { address: DynamicGlobalAddress, method_name: String, args: ManifestValue }`

This calls a role-assignment module method.

Instruction: `CallDirectVaultMethod`

Signature: `CallDirectVaultMethod { address: InternalAddress, method_name: String, args: ManifestValue }`

This calls a method directly on a vault, bypassing standard access controls.

6. Address operations

Instruction: `AllocateGlobalAddress`

Signature: `AllocateGlobalAddress { package_address: PackageAddress, blueprint_name: String }`

This creates global address reservation.

7. Metadata and role operations

Instruction: `SetMetadata`

Signature: `SetMetadata { address: ComponentAddress, args: ManifestValue }`

This sets metadata for a component.

Instruction: `RemoveMetadata`

Signature: `RemoveMetadata { address: ComponentAddress, args: ManifestValue }`

This removes metadata from a component.

Instruction: `LockMetadata`

Signature: `LockMetadata { address: ComponentAddress, args: ManifestValue }`

This locks component metadata to prevent further changes.

Instruction: `SetComponentRoyalty`

Signature: `SetComponentRoyalty { address: ComponentAddress, args: ManifestValue }`

This sets royalty configuration for a component.

Instruction: `LockComponentRoyalty`

Signature: `LockComponentRoyalty { address: ComponentAddress, args: ManifestValue }`

This locks component-royalty configuration.

Instruction: `ClaimComponentRoyalties`

Signature: `ClaimComponentRoyalties { address: ComponentAddress, args: ManifestValue }`

This claims accumulated royalties for a component.

Instruction: `SetOwnerRole`

Signature: `SetOwnerRole { address: ComponentAddress, args: ManifestValue }`

This sets the owner role for a component.

Instruction: `LockOwnerRole`

Signature: `LockOwnerRole { address: ComponentAddress, args: ManifestValue }`

This locks the owner-role configuration.

Instruction: `SetRole`

Signature: `SetRole { address: ComponentAddress, args: ManifestValue }`

This sets a role for a component.

8. Component-creation operations

Instruction: `CreateAccessController`

Signature: `CreateAccessController { args: ManifestValue }`

This creates a new access-controller component.

Instruction: `CreateIdentity`

Signature: `CreateIdentity { args: ManifestValue }`

This creates a new identity component.

Instruction: `CreateIdentityAdvanced`

Signature: `CreateIdentityAdvanced { args: ManifestValue }`

This creates a new identity component with advanced configuration options.

Instruction: `CreateAccount`

Signature: `CreateAccount { args: ManifestValue }`

This creates a new account component.

Instruction: `CreateAccountAdvanced`

Signature: `CreateAccountAdvanced { args: ManifestValue }`

This creates a new account component with advanced configuration options.

9. Cleanup operations

Instruction: `DropNamedProofs`

Signature: `DropNamedProofs`

This drops all named proofs.

Instruction: `DropAllProofs`

Signature: `DropAllProofs`

This drops all proofs (named and auth zone).

5.4. Module: Kernel

Nodes and substates

In Radix, everything in the kernel is a node. This includes blueprints, packages, components, and anything related to blockchain state.

Each node can have their own state. This state is generally stored in substate structures.

Nodes can be stored in two different types of devices: the store and the heap.

Substate devices

The heap device is used during transactions, and there is an invariant that ensures that there are no live references to nodes on the heap when the transaction ends. It has the following structure,

```
pub struct Heap {
    nodes: NonIterMap<NodeId, NodeSubstates>,
}
```

where `NodeId` is just a wrapped `[u8; 30]`, while the `NodeSubstates` structure is a `BTreeMap` that maps partitions to substates:

```
pub type NodeSubstates = BTreeMap<PartitionNumber, BTreeMap<SubstateKey,
    IndexedScryptoValue>>;
```

Each substate is keyed by a `SubstateKey` (which is either a `u8`, a `Vec<u8>`, or a `([u8; 2], Vec<u8>)` depending on the type of key being used) and stored as an `IndexedScryptoValue`:

```
pub struct IndexedScryptoValue {
    bytes: Vec<u8>,
    references: Vec<NodeId>,
    owned_nodes: Vec<NodeId>,
    scrypto_value: RefCell<Option<ScryptoValue>>,
}
```

This implies that substates can hold references to other nodes, which is important because, as previously mentioned, there is an invariant that ensures that there are no live references to other nodes when the transaction ends.

Therefore, it is important that at the end of a transaction, there are no substates on the heap with nodes in their `owned_nodes` or `references` vectors, while those nodes themselves are also on the heap, as that would mean that there is a live reference to this node.

The store device is where permanent substates live. This includes globalized component and blueprint states as well as globalized nodes themselves. The store device keeps track of nodes that are created in the current transaction as well as transient substates (that is, substates that are only intended to be alive for the duration of a transaction).

```
pub struct Track<'s, S: SubstateDatabase, M: DatabaseKeyMapper + 'static> {
    /// Substate database, use `get_substate_from_db` and
    `list_entries_from_db` for access
    substate_db: &'s S,

    tracked_nodes: IndexMap<NodeId, TrackedNode>,
    force_write_tracked_nodes: IndexMap<NodeId, TrackedNode>,
    /// TODO: if time allows, consider merging into tracked nodes.
    deleted_partitions: IndexSet<(NodeId, PartitionNumber)>,

    transient_substates: TransientSubstates,

    phantom_data: PhantomData<M>,
}
```

The `substate_db` field refers to a `SubstateDatabase`, which is a database structure that stores globalized and otherwise permanent blockchain states. During a transaction, this database is used as a starting point, while at the end of the transaction, the database is updated with any newly created globalized nodes as well as any permanent changes to the blockchain state.

Substate structures

The actual data structures that are used to store specific types of blockchain data are what we will refer to as substate structures. For example, the substate structure to track the validator rewards looks like this:

```
pub struct ValidatorRewardsSubstate {
    pub proposer_rewards: IndexMap<ValidatorIndex, Decimal>,
    pub rewards_vault: Vault,
}
```

Where the `proposer_rewards` map tracks the actual reward amounts per validator and the `rewards_vault` is the vault that contains the rewards themselves.

These substates are generally part of a globalized component, or a blueprint. In this case, the `ValidatorRewardsSubstate` is part of the global consensus-manager component, which handles consensus for the Radix blockchain. The globalized component or blueprint in this instance is referred to as a node in the Radix engine's kernel.

Substates are not directly stored as part of the node, however. Instead, nodes can have modules. In the above example, the `ValidatorRewardsSubstate` would be part of the consensus-manager component's SELF module. The module itself contains fields, which is where substates are stored.

As an example, the substate above is accessed as follows:

```
// Zellic: Open a handle to the `SELF` module, and access the
    `ValidatorRewards` field
let rewards_handle = api.actor_open_field(
    ACTOR_STATE_SELF,
    ConsensusManagerField::ValidatorRewards.into(),
    LockFlags::MUTABLE,
)?;

// Zellic: Read the substate that is stored within this field.
let mut rewards_substate = api

    .field_read_typed::<ConsensusManagerValidatorRewardsFieldPayload>(rewards_handle)?
    .fully_update_and_into_latest_version();
```

The node that ends up storing the substate is in turn converted to an `IndexedScryptoValue`, which will then track any other nodes that this node references or owns.

Manifests

In the Radix engine, every transaction starts from a manifest. The manifest describes the list of actions that will occur in the transaction as well as how resources are intended to move between components, payments of transaction fees, and so forth.

Once the manifest ends up in the Radix engine, the engine initially verifies that every component that is referenced in the manifest actually exists in the global blockchain state. This is done in three steps:

1. If a referenced component is a component that is within the `ALWAYS_VISIBLE_GLOBAL_NODES` list, then this component is allowed to be referenced in the manifest. This list includes all the native blueprints.

2. If a referenced component is a global virtual component, then it is allowed. In Radix, components of the accounts and identity blueprints are considered global virtual. These are allowed by default because even if no such components exist on the chain at the time of the transaction, they can be created in the future, and thus other components should still be allowed to interact with them.

3. If neither of the above cases are true, then the global blockchain state is accessed to ensure that this node actually exists. This would then mean that this component is a glob-

alized component (for example, a user-created blueprint that has been globalized previously) and thus can be accessed through a transaction manifest.

Once the manifest passes these checks, the instructions in the manifest and the list of referenced components are passed into the native transaction-processor blueprint's `run()` function. This function handles executing all the instructions in the manifest.

The call to this `run()` function triggers the creation of a new call frame.

Call frames

A call frame is created for each new function/method call made in a transaction. The `CallFrame` structure is as follows:

```
pub struct CallFrame<C, L> {
    /// The frame id
    depth: usize,

    /// Call frame system layer data
    call_frame_data: C,

    /// Owned nodes which by definition must live on heap
    owned_root_nodes: IndexSet<NodeId>,

    /// References to non-GLOBAL nodes, obtained from substate loading, ref
    counted.
    /// These references may NOT be passed between call frames as arguments
    transient_references: NonIterMap<NodeId, TransientReference>,

    /// Stable references points to nodes in track, which can't moved/deleted.
    /// Current two types: `GLOBAL` (root, stored) and `DirectAccess`.
    /// These references MAY be passed between call frames
    stable_references: BTreeMap<NodeId, StableReferenceType>,

    next_handle: SubstateHandle,
    open_substates: IndexMap<SubstateHandle, OpenedSubstate<L>>,
}
```

In the above structure, the `owned_root_nodes` set contains all nodes that are owned by the current call frame (which essentially means that the node was created within this call frame). The nodes in this set must be on the heap, which means that these nodes are not globalized. This includes nodes created as part of blueprints and components, such as vaults, buckets, and so on.

The `transient_references` map also contains nodes that are only on the heap, but instead of the nodes here being created within this call frame, these nodes are instead loaded when other substates are loaded. The `TransientReference` structure is as follows:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord)]
pub struct TransientReference {
    ref_count: usize,
    ref_origin: ReferenceOrigin,
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord)]
pub enum ReferenceOrigin {
    FrameOwned,
    Global(GlobalAddress),
    DirectlyAccessed,
    SubstateNonGlobalReference(SubstateDevice),
}
```

The `stable_references` map contains nodes that are stored in the `Store` substate device, meaning they are part of the permanent blockchain state and can never be removed. The `StableReferenceType` is an enum that describes the node. These nodes can either be `Global` or `DirectAccess`, and these nodes can either be created within the current call frame (and subsequently globalized) or loaded through other substates.

The `open_substates` structure tracks the current open substates within this call frame. All open substates must be closed when the call frame is dropped.

When a new call frame is created for a function/method call in a transaction, the arguments to the function/method call are passed in as an `IndexedScryptoValue`. This tracks all the references and owned nodes of the arguments. These are then moved into the new call frame, and so the call frame starts off with the required references needed to execute the function/method call.

When a call frame is dropped, the return value of the function/method call is used to do the same thing as above but in reverse. The return value is an `IndexedScryptoValue`, which contains any references and owned nodes that must be moved back to the parent call frame.

It is important to note that transient references cannot be passed between call frames. This invariant exists in the code base because it prevents the need for adding additional checks to ensure the references are still alive and have not been dropped when the references are passed to another call frame.

Node visibility

Nodes are identified by unique node IDs. A node ID may look like the following:

```
f8305dfa7742a66ce8c8c299cfdeb6f597b44dd381fb28cb808a72e2cf10
```

A user-defined blueprint potentially has the ability to spoof nodes, which essentially means that they can construct a component with the exact same node ID as another one, and then try to access it. How does Radix prevent this?

Radix has a concept known as node visibilities. Whenever a node is accessed (for example, a substate is opened for reading), the engine makes sure that the node in question is visible in the current context. This means the node has to fall in one of two categories:

1. The node must be visible in the current call frame (that is, existing in the `owned_root_nodes`, `transient_references`, or `stable_references`).
2. If the node is not visible in the current call frame, it must be one of the `ALWAYS_VISIBLE_GLOBAL_NODES`.

These rules ensure that it is impossible for nodes to be spoofed and accessed through other nodes that do not have permissions for it.

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to the Radix mainnet.

During our assessment on the scoped Radix component, we discovered six findings. No critical issues were found. Three findings were of medium impact, two were of low impact, and the remaining finding was informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.