Security Audit Report

Soroban Environment Audit Stellar

Delivered: February 19, 2025

runtime verification

Table of Contents

Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an 'as-is' basis. You acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Blockchain technology is still a nascent software arena, and any related implementation and public offering carries substantial risk.

Finally, the possibility of human error in the manual review process exists, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Executive Summary

The Stellar Development Foundation (SDF) engaged Runtime Verification Inc. to conduct a security audit of the Soroban smart contract platform. The objective was to review the logic and implementation of critical components and identify any issues that could cause erroneous or undefined behavior, potentially leading to exploitation or malicious interaction with the Stellar network.

The audit was conducted over a period of approximately 10 calendar weeks, concluding on December 23, 2024. It focused on analyzing the following accepted Core Advancement Proposals (CAPs): **CAP-0051**, **CAP-0053**, **CAP-0054**, **CAP-0055**, **CAP-0056**, **CAP-0058**, **CAP-0059**, and **CAP-0060**.

Given the extensive and complex nature of Soroban's codebase, a comprehensive approach was adopted to ensure the highest guarantees within the allocated timeframe. The audit encompassed two primary areas: a thorough code review of the specified CAPs, prioritized by their criticality, and dedicated fuzz testing using a variety of tools and configurations.

The Soroban codebase is well-structured, adhering to best practices and containing informative documentation that clarifies complex invariants.

The audit uncovered a problem (Panic discovered in `expr` fuzzing target) through the fuzzing campaign that could lead to an internal host error.
Furthermore, as an informative finding, the crate dependencies include a crate with a use-after-free, which is an optional dependency and easy to eliminate.

Audit Scope & Goals

The goal of this audit was to assess the security and correctness of Soroban's new features and updates introduced through a series of Core Advancement Proposals (CAPs) since the previous audit for Protocol 21, conducted by Veridise and completed on January 3, 2024. Runtime Verification Inc. and the Stellar Development Foundation (SDF) aligned on an approach designed to maximize both the coverage and depth of the audit within the allocated timeframe while addressing the specific risks associated with these recent changes.

The audit aimed to:

1. Identify Critical Vulnerabilities: Analyze the code implementation to detect errors or security flaws that could compromise the stability or functionality of the Stellar network.
2. Evaluate Compliance with Protocol Requirements: Ensure the new CAPs have been implemented as intended and adhere to the established protocol specifications.
3. Highlight Informative Findings: Provide additional recommendations to improve the safety, efficiency, or readability of the codebase.

Scope

The target code for this audit resides in two GitHub repositories of the Stellar Foundation, and the version to audit has the commit hashes given here, tagged as `22.0.0-rc1.1`:

- https://github.com/stellar/rs-soroban-env, at commit `f0bc81b861efbb07f9406790cad736db90147e12`
- https://github.com/stellar/rs-stellar-xdr, at commit `72e523004b5906eb1829990f9b14d2f0fa3018f0`

This audit is focused on the Soroban-related changes introduced through the following Core Advancement Proposals (CAPs): CAP-0051, CAP-0053, CAP-0054, CAP-0055, CAP-0056, CAP-0058, CAP-0059, and CAP-0060. These CAPs introduced new functionalities, including cryptographic support, enhanced host functions, and optimizations crucial for Soroban's evolution.

The review encompassed the code contained in the Soroban environment and Stellar XDR repositories as provided by the client, with an emphasis on changes and updates since the previous audit. The primary areas of focus included:

- Implementation of CAP Changes: Review of new host functions, metering, error handling, and cryptographic features to ensure proper integration and functionality.
- Security Concerns Identified in CAP Documentation: Examination of potential attack vectors, such as denial-of-service vulnerabilities and unchecked arithmetic, to validate the proposed mitigations.
- Backward Compatibility and Stability: Assessment of whether the new changes maintain compatibility with existing Soroban contracts and Stellar's broader ecosystem.

By centering the audit on the most recent updates and enhancements, the objective was to provide Stellar Network with actionable insights to ensure the security and reliability of Soroban’s evolving smart contract platform.

Methodology

The codebase, being both large and complex, required a multifaceted approach to its review and audit. While the primary focus was on specific CAPs, a broader understanding of the codebase was essential for contextual analysis. While prioritizing code segments directly related to the CAPs, we also identified potential areas for future audits.

To gain a thorough understanding of the codebase, we initiated a series of in-depth code walkthroughs with the client and reviewed existing documentation, leading to a concise system description through targeted code reviews. A deep dive into the XDR format and internal data types, including conversions between them, was crucial for both code review and subsequent fuzzing efforts. We documented the macro system that defines the host’s data types and described macro relationships involving datatypes and host function macros through graphs and text. This documentation aims to guide new developers in utilizing macros within the system. Regular meetings with the client, held weekly or more frequently, facilitated ongoing discussions and addressed emerging issues.

For each CAP, we began by thoroughly reviewing documentation to grasp its intent and design. In-depth code analysis, including surrounding code, was performed to identify potential issues such as common Rust errors, type conversion problems, and logical errors. GitHub discussions related to each CAP were examined to understand existing concerns. A comprehensive summary was prepared for each CAP and internally reviewed by a second auditor to verify accuracy and discuss findings. Additional meetings with domain experts were scheduled to gain deeper insights into specific CAPs.

The fuzzing process involved a comprehensive evaluation of existing fuzz targets and property tests, ensuring their functionality and identifying those deserving of increased attention. Prioritizing fuzzing efforts on conversions between XDR and internalized types, we verified round-trip conversions and consistency in data ordering. Existing fuzz targets and property tests were adapted to leverage honggfuzz for genetic fuzzing, and a data generator was developed to prevent invalid data input.

To identify crashes related to linear memory usage, we created a harness to simulate a VM environment with linear memory for fuzzing host operations. Simple read-write operations involving linear memory were targeted, and budget constraints were varied to test error handling. Read operations on corrupted linear memory data were fuzzed to ensure robustness. Deep nested call fuzzing involved simulating complex call stacks to test error propagation and rollback mechanisms, verifying correct behavior in scenarios involving budget exhaustion during error handling, and ensuring the accuracy of the authentication callstack.

To further enhance the codebase's reliability and security, we recommend future investigations into error code analysis, and a review of the authorization and authentication module. Error code analysis would scrutinize the use of the Result type in host operations to identify potential error conversion issues, detect instances of error suppression, and consider developing a tool to automate this analysis on the lowered representation. A review of the authorization and authentication module would involve refactoring the monolithic module for better modularity and addressing potential data duplication issues.

Description of the System

Repository `rs-stellar-xdr`

The code in `rs-stellar-xdr` is the crate `stellar-xdr`. It provides definitions and utilities to process and convert binary data for the Stellar ledger in XDR format.

The library provides a `curr` and a `next` variant of the Stellar ledger datatypes, referring to particular versions (commit hashes) of the `stellar-xdr` repository. Most code in `rs-stellar-xdr` is auto-generated from XDR files in the latter repository.

The crate also contains a small CLI application which can convert between XDR binary data and a json representation, as well as compare different XDR files and guess the type of XDR-encoded data.

Repository `rs-soroban-env`

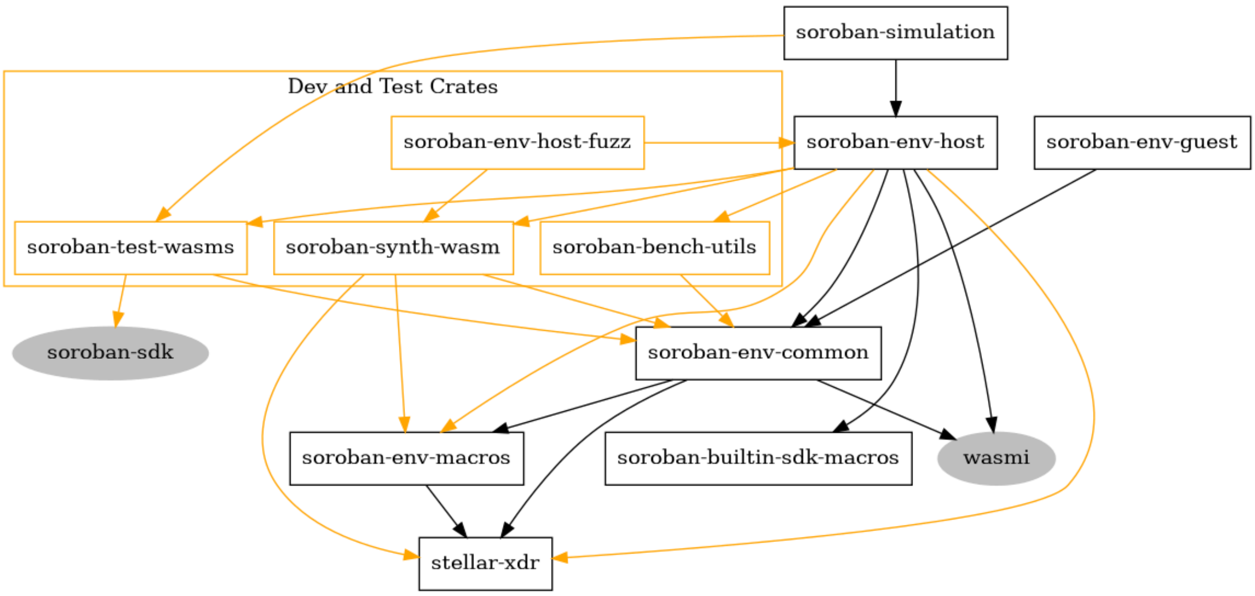
The code in `rs-soroban-env` provides code to implement the environment in which Stellar smart contracts are executed and interact with the ledger data in transactions.
The code is organised in a number of different crates:

- `soroban-env-host`: code for the management of guest code execution within a transaction, including the authorization of contract functionality, as well as interfacing to the Wasmi interpreter for the actual execution, budgeting the execution resources, and managing transactional access to ledger data.
- `soroban-env-guest`: an interface for the guest code to program against. This code can be compiled in "host mode", such that it is directly linked to the previous crate for tests, or in "guest mode", where calls go through a Wasm import of host functions;
- `soroban-env-common`: data types and utilities for data that is accessed by both the guest code and the host code;
- `soroban-env-macros`: macros used extensively to generate the code in `soroban-env-common` that relates to host functions and type conversions between Stellar XDR types and the host's internal `Val` type (and its subtypes).
- `soroban-simulation`: provides the functionality for running transactions as a "pre-flight", i.e., in "recording mode" to collect the set of ledger data read or written as well as the required authorizations.

Other crates in the repository are used for testing and for benchmarking and calibrating the cost attributed to each operation.

- `soroban-bench-utils` : types and utility functions for cost calibration
- `soroban-env-host-fuzz` : test crate to run fuzz tests with generated Wasm modules
- `soroban-synth-wasm` : test crate to provide a Wasm module generator
- `soroban-test-wasms` : pre-compiled Wasm for a number of test and example programs (originally from `soroban-sdk`)
- `soroban-builtin-sdk-macros` : (simplified versions of) annotation macros from `soroban-sdk` for use in built-in contracts in `soroban-env-host` .

The full crate dependency graph is shown here, with test and development tool dependencies in orange.



Notable external dependencies are the `soroban-sdk` crate (for writing soroban smart contracts) and the `wasmi` crate (providing the integrated Wasm interpreter).

Functionality

Soroban Environment is the codebase that implements the host for smart contracts on the Stellar network. Smart contracts are Wasm bytecode on the ledger which may import and use host functions.

Stellar provides a rich environment of host functions, and in turn restricts the data types within the smart contracts to just 64-bit integers.
The design goal is that the smart contracts won't have to contain any code for encoding their data, because common data types and operations are more efficiently provided by the host. A contract's data will thus exist as a set of *host objects* addressed from the smart contract code by way of *handles*.

The data on the host that smart contracts operate on are `Val` s, a tagged union defined in `soroban-env-common` with 8 bytes of payload. `Val` encodes `Bool` , different number and symbol types as well as handles to host objects of more complex type and/or larger size. Values from the ledger are XDR-encoded `SCVal` s which are converted to and from `Val` , also creating host objects where necessary (generally any type prefixed with `SC` indicates a relationship to XDR data). Macros are used extensively to define this system of datatypes and conversions. A description of the macros and a diagram of the resulting data types and their relationship can be found in the [Appendix: Data Type Macros and Diagram for `rs-soroban-env`].

The Stellar-Soroban `Host` is a structure gathering all state components relevant to smart contract execution within a transaction. Most notably, this includes a *call stack* (`context_stack`) of calls between contract code and host code (the former managed in distinct Wasm `Vm` s associated with one stack frame each), a set of *host objects* referred-to and exchanged by the guest code, and an `AuthorizationManager` which manages the code's permissions (and also authentications) according to the Stellar authorization model, and the `Budget` of the running transaction.

All operations on the host are "budgeted", meaning that each operation deducts an associated cost from the host's budget, and the transaction fails if an insufficient budget was supplied. Budget costs are standard values per operation, with the costs determined by benchmarking, and reflecting worst-case execution costs for memory allocation and CPU instructions. Costs are compiled into the host code.

Because of the budget and the data model using host objects for any data that cannot be represented in 56 bits, most of the code in `soroban-env-host` needs the `Host` as a mutable reference.

The host can execute transactions in *recording mode* to determine the required budget, collect the read and write set of the transaction (a ledger storage `Footprint`), and determine what authorisations are required for it to succeed. This functionality is packaged in the `soroban-simulation` crate.

When a transaction is submitted for actual validation and execution, evidence of authorization as per the collected authorization requirements have to be provided, and the transaction cannot access or modify any data outside the recorded storage `Footprint` (including the distinction between read and write set).

Host functions

A large catalog of *host functions* is provided by the smart contract environment. The main motivation is to avoid having to implement low-level conversions from bytes to more specific data types in smart contract code, and to avoid (other) duplicated code for common types such as maps or vectors.

All host functions are described in `soroban-env-common/env.json`, using a simple json format that groups functions by module, includes function name and a short call name (to save import table space), and type information about the arguments and returned data, as well as a description string.

Code related to host functions is generated using macro `call_macro_for_all_host_functions` more often than not. This macro takes another macro as an argument and calls it with the data of `env.json`. Typical uses of the macro can be found in `soroban-env-guest/src/guest.rs` (creating `extern C` stub declarations for all host functions), or in `soroban-env-host/src/vm/func_info.rs` (to generate Wasmi linker information).

Mechanisms to provide host functions to guest code

`soroban-env-host` implements the host functions in the `VmCallerEnv` trait implementation.

Smart contract developers can compile and run the smart contract (guest) code in "host/test mode", which means that the code implementing the host function is called directly through the `Env` trait implementation derived from the `VmCallerEnv` one.

In `soroban-env-guest`, host functions exist as extern declarations so that guest code can be written and compiled in "guest/production mode" which calls these host functions via Wasm and through the host interface. In this "guest/production mode", the host functions have to be supplied to Wasm code that runs within a `Vm` by way of a Wasmi interpreter. The host functions can be imported into this Wasm code, based on the Wasmi `Linker` functionality, which will dispatch function calls to the host function implementation.

For this purpose, the list of `soroban-env-host::vm::func_info::HOST_FUNCTIONS` contains all host functions with their import names (module and short name, used in Wasm imports), long names (which are identical to the ones in `VmCallerEnv`), arity information, and a `wrap` function which adds them to a `wasmi::Linker`. For all host function calls from Wasm, the linker will dispatch to an implementation with the same long name in `vm::dispatch` which decodes all arguments from `i64` to `Val` and then calls the `VmCallerEnv` implementation. All of these implementations are generated using the `call_macro_with_all_host_functions` with different helper macros.

Since the change in `CAP-55`, only host functions which are *actually imported* in the loaded Wasm module are added to the Wasm linker. This addition to the linker happens when the module is added to the *module cache* (`CAP-56`), therefore the linker contains all host functions used in any of the cached modules.

Ledger Storage

The `Host` contains a `Storage` object to control ledger data access. Any data read or written by guest code during a transaction must be in the `Storage` of the host.

`Storage` holds ledger data in a `Map` (implemented as a sorted array) keyed by `LedgerKey`. Amoung the `LedgerKey` variants, mainly `ContractData` and `ContractCode` are relevant to this storage access within a transaction, other ledger keys are not supposed to be used in the `Storage` module (see `Storage::check_supported_ledger_key_type` in `storage.rs`, otherwise it is an *internal* error).

During "pre-flight", comprehensive ledger data is sourced from a snapshot. Pre-flight execution then records which data from the ledger snapshot is actually accessed, in a `Footprint` data structure distinguishing `ReadOnly` and `ReadWrite` accesses.

For actual execution, the `Footprint` is used to initialise the `Storage` cell of the `Host` for the transaction. All write accesses to ledger data are checked to have the `ReadWrite` access registered in the `Footprint`. Access to the stored data is further checked with the help of the *contract address* part of the `LedgerKeyContractData`, to ensure a contract does not read another one's data.

Authorization

The Soroban authorization subsystem provides a way for *addresses* (accounts or contracts) to authorize a sequence of nested function calls and contract creations. A tree of `AuthorizedInvocation`s which represents the nested call sequence is provided with a transaction and checked during execution whenever contract code calls the `address.require_auth` host function (meaning that the given `address` needs to have authorized the current function call context).

Authorizations are *consumed* when they match with a `require_auth` call, repeated function calls need individual authorizations for each call. The authorization record that is checked includes the arguments of the function call (or contract creation with constructor). During pre-flight, the required authorizations are *collected* in order to provide these authorizations with the transaction request.

Code Review Discussion and Findings

This section outlines the results of our manual code review, where our team conducted a thorough analysis of the target codebase. The focus was on identifying potential vulnerabilities, logical errors, and areas of improvement, while ensuring that the implementation adheres to best practices and intended design. Findings are categorized by severity and include recommendations for remediation.

CAPs to focus on

A number of specific changes, described in CAPs (Core Advancement Proposals), were pointed out by the client as a particular focus for the audit. In this section, we analyse the functionality for each of these CAPs in some detail.

- **CAP-51** - Support for `secp256r1`
- **CAP-53** - TTLs extensible separately for contract instance and code
- **CAP-54** - refined module cost (determined during contract upload, charged when instantiating)
- **CAP-55** - Only host fns used by a contract are added to the linker.
- **CAP-56** - Module cache used when same contract is invoked multiple times during a Soroban tx.
- **CAP-58** - Support constructors to initialize contract during upload.
- **CAP-59** - BLS elliptic curve support
- **CAP-60** - Wasmi register engine (planned to use it in Protocol 23)

CAP-51: Smart Contract Host Functionality: Secp256r1 Verification

This CAP proposed adding host functions for SEC2 secp256r1 / NIST P-256 Elliptic Curve Digital Signature Algorithm (ECDSA).

Previous to CAP-51, SEC2 secp256k1 ECDSA was possible through the host function recover_key_ecdsa_secp256k1. After CAP-51, ECDSA is also possible through verify_sig_ecdsa_secp256r1. `verify_sig_ecdsa_secp256r1` uses a different curve then `recover_key_ecdsa_secp256k1` and also a different verification method (verify instead of recover). `verify_sig_ecdsa_secp256r1` depends on the third party crates p256 and ecdsa. While these crates seem robust and state-of-the-art, the documentation indicates that no independent audit has been conducted, which does mean usage of these crates includes some amount of risk and increases the trust base.

To ensure safe implementation and use of `secp256r1` ECDSA, known vulnerabilities and common pitfalls were analysed.

Signature Malleability

A known pitfall with ECDSA is signature malleability attacks. This has been recognised and is protected in code by restricting the signature `s` value to low values, meaning the corresponding reflected signature with high `s` will be rejected if attempted to be used.

Randomness, and Hash Function Strength

ECDSAs such as `secp256r1` rely on randomised One Time Secret Number (OTSN) generation, and hash functions. It is critical to the security of private keys that the OTSN is never reused and is generated sufficiently randomly each time. "Sufficiently randomly" means that an attacker would not be able to gain any knowledge to be able to predict the random number, or any bits of the number, better than a random guess. It is also critical that the hash function used is sufficiently resistant to collisions and preimage recovery. If either of these conditions are not met, then it is possible for an attack to solve a set of simultaneous equations with at least two signatures to recover the private key.

It is the responsibility of the user providing the signatures to the newly implemented host functions to ensure the OTSN is indeed fresh and sufficiently random. The SDK provides SHA256 and Keccak256 as hash functions, and these are considered sufficiently strong hash functions. However if a user was interacting with the host functions directly, it is their responsibility to use a sufficiently strong hash function. In the event that sufficiently random values cannot be produced, there is Deterministic Digital Signature Scheme (DDSS) available, however it should be noted that this is susceptible to fault attacks *if* they are viable.

Side Channel Attacks

`secp256r1` is considered resistant to side channel attacks, however it is important to make sure all implementations and libraries are current to avoid exploits on previous implementations. The currently chosen implementation of `secp256r1` (p256) has an algorithm for constant time arithmetic, and is not considered susceptible to side channel attacks. Functions in the library that do not guarantee to implement constant time arithmetic (`*_vartime`) are not used.

Fault Attacks

As mentioned above, DDSS (and ECDSA that uses DDSS including `secp256r1`) may be susceptible to fault attacks [1] if fault attacks are considered part of the threat model. The need for DDSS generally stems from a lack of computational resources to generate sufficiently random numbers. It is unlikely that a user or application should need to use DDSS, however, but if they do they should have consideration for this attack vector.

1. Barenghi, A., Pelosi, G. (2016). A Note on Fault Attacks Against Deterministic Signature Schemes (Short Paper). In: Ogawa, K., Yoshioka, K. (eds) Advances in Information and Computer Security. IWSEC 2016. Lecture Notes in Computer Science(), vol 9836. Springer, Cham. https://doi.org/10.1007/978-3-319-44524-3_11 ↩

CAP-53: Separate host functions to extend the TTL for contract instance and contract code

This CAP proposed to add host functions to extend the time to live (TTL) for smart contract instance (storage and code pointer) and smart contract code separately through newly added host functions.

Previous to CAP-53 the only methods through Soroban[1] to extend the TTL for either the contract instance or the contract code were the Soroban host functions extend_current_contract_instance_and_code or extend_contract_instance_and_code which simultaneously extend the contract instance and code if a valid TTL is provided. CAP-53 extends the Soroban host functions with extend_contract_instance_ttl and extend_contract_code_ttl which exclusively extend the ttls the contract instance and code, respectively.

Stellar Storage

Storage on the Stellar network is either TEMPORARY or PERSISTENT (note: contract instance and code is PERSISTENT) that differ in behavior dependent on the TTL. If the TTL is above 0 both TEMPORARY and PERSISTENT storage are considered LIVE and are accessible. If the TTL drops to 0 then both TEMPORARY and PERSISTENT are not accessible, however PERSISTENT storage is recoverable and is in ARCHIVED state while TEMPORARY storage is not recoverable and is in DEAD state. PERSISTENT storage cannot be in DEAD state, and TEMPORARY storage cannot be in ARCHIVED state. A summary of states and their accessibility A and recoverability R:

	LIVE	DEAD	ARCHIVED
TEMPORARY	A	!A & !R	-
PERSISTENT	A	-	!A & R

Contract Instance and Code

When a contract is deployed it is deployed in PERSISTENT storage as an instance (`ScContractInstance`), where an instance is a pointer to a ledger entry that stores the Wasm bytecode (`executable`) and a pointer to the ledger entry that stores data storage map (`storage`) which may also be omitted if no data is to be stored. The pointer to the bytecode is the SHA256 hash of the Wasm bytecode, and multiple deployed smart contracts can point the same bytecode. Here is a diagram to illustrate the entries on the Stellar Ledger (the abstract layout is assumed to be contiguous for simplicity):

```
X is the size of a LedgerEntry
WASM is the Wasm bytecode of the smart contract
WASM.len is the length of the Wasm bytecode of the smart contract
DATA is the instance storage entries
DATA.len is the length of the instance storage entries
TTL_I is the TTL for the contract instance
TTL_C is the TTL for the contract code

           |-----------LedgerKey----------|-----------LedgerEntry---------|
           | ...prior keys...             | ...prior entries...           |
         ┌─ | InstanceStorageKey           | Hash(WASM)                    |
TTL_I ─┤  | InstanceStorageKey + 1       | DATA[0]                       | ─┐
         │  | ...                          | ...                           |  ├── Possibly
         └─ | InstanceStorageKey + DATA.len | DATA[DATA.len - 1]            | ─┘   empty
         ┌─ | Hash(WASM)                   | WASM[0..X]                    |
TTL_C ─┤  | ...                          | ...                           |
         └─ | Hash(WASM) + WASM.len - X    | WASM[(WASM.len-X)..(WASM.len)] |
           | ...subsequent keys...        | ...subsequent entries...      |
```

When a contract extends the TTL of their contract code they increase the TTL of all the entries storing the code pointed to by the hash. When a contract extends the TTL of the instance they increase the TTL of the hash of the Wasm and of all entries in the instance storage. Since multiple contracts can point to the same contract code, prior to CAP-53 it could be the case that the TTL of the contract code is being updated inefficiently by many contracts. This inefficiency comes from the unnecessary frequency of extension to the TTL, and that the updates to the code are expensive as they span across multiple ledger entries. CAP-53 allows users to separately update the instance and the code which will mean that the inefficient updates to the code can be avoided.

CAP-53 State Analysis

The possible states available for the contract instance and code are {LIVE, ARCHIVED} $\times$ {LIVE, ARCHIVED} currently. CAP-53 offers no expansion to the possible states since there is the possibility to extend TTL via Stellar operation `ExtendFootprintTTLOp` [1] which means it has already been possible to achieve the individual extensions (just not inside Soroban). Any attempted access to ARCHIVED storage will result in failure during preflight construction of the footprint.

All host functions that extend the TTL of storage call `extend_ttl`, see Appendix: `extend_ttl` Sequence Diagram for sequence diagram of `extend_ttl` from a call to `extend_contract_instance_ttl`.

1. External to Soroban, the TTL for a ledger entry (including contract instance, and code) can be increased directly through Stellar operation ExtendFootprintTTLOp. ↩ ↩[2]

CAP-54: More Granular Cost Model for Module Loading

This CAP refined the cost model for Soroban smart contract execution by introducing separate cost categories for parsing and instantiating WebAssembly (Wasm) modules.

This change was implemented in commit 41b4ee3, which also contains the implementation of a module cache (CAP-56). Prior to this CAP, the cost model for Soroban smart contract execution applied a single fee for the instantiation of the Virtual Machine (VM) to run Wasm-based contracts. However, this model did not adequately distinguish between the costs of parsing the Wasm module (which happens once per transaction) and the costs associated with instantiating the contract (which can occur multiple times during contract execution). This change enables a more accurate metering of resource usage, aligning costs with computational complexity.

Before, parsing and instantiating a Wasm module were handled as a single unit under the VM instantiation cost. Now, they are separated into two distinct categories:

1. Parsing Costs (C_{parse}):
    - Applied once per transaction for each unique Wasm module.
    - Covers the computational effort required to decode and validate the module.
    - Parsing costs do not depend on how often the module is invoked.

2. Instantiation Costs ($C_{\text{instantiate}}$):
    - Applied dynamically for each invocation of a module.
    - Reflects the effort of creating a runtime instance (VM) from a parsed module.

These refinements enable:

- Fairer fee assessments aligned with actual resource consumption.
- Improved efficiency for transactions involving repeated contract invocations.
- Integration with module caching (CAP-56) for enhanced scalability.

When a VM instance is created for contract execution, it requires that host functions be linked. This linking process depends on whether the VM is created from:

1. A cached parsed module: In this case, host functions are extracted from the module cache.

2. A newly parsed module: Host functions are resolved directly from the module being instantiated.

Granular Cost Model

Total Transaction Cost

The total cost of a transaction is:

$$C_{\text{total}} = \sum_{m=1}^{M} C_{\text{parse},m} + \sum_{n=1}^{N} C_{\text{instantiate},n} + C_{\text{execute}}$$

Where:

- $C_{\text{parse},m}$ is the cost of parsing the m-th Wasm module.
- $C_{\text{instantiate},n}$ is the cost of instantiating the module during the n-th invocation.
- C_{execute} is the runtime cost for contract execution.
- M is the number of unique Wasm modules parsed in the transaction.
- N is the total number of instantiations across all modules in the transaction.

Parsing Costs: C_{parse}

Parsing costs reflect the computational effort of decoding and validating a Wasm module. These costs occur only once for each unique module, either during the first upload or if the module is not in the cache.

The formula for parsing cost is:

$$C_{\text{parse},m} = \begin{cases} k_{\text{parse}} \cdot S_m & \text{if first time parsing (Recording Mode)} \\ k_{\text{parse}} \cdot X_m & \text{if retrieved from cache (Enforcing Mode)} \end{cases}$$

where:

- S_m is the size of the m-th module in bytes.
- X_m is the complexity of the m-th module (such as the number of functions, imports, and data segments).
- k_{parse} is the scaling factor that defines the unit cost of parsing operations.

When a module is parsed for the first time, its size determines the parsing cost, as it requires full decoding and validation. This operation occurs only once for each unique module during the first upload.

Once the module is parsed and cached, the parsing cost is determined by its complexity. This complexity takes into account factors such as the number of functions, imports, and other attributes that influence the module's runtime performance.

Instantiation Costs: $C_{\text{instantiate}}$

Instantiation costs are dynamically charged whenever a contract is invoked during enforcing mode. Each invocation incurs a cost based on the following:

- Invocation Frequency (N): Number of times the module is instantiated.
- Runtime Complexity (Y): Memory allocations and other runtime requirements.

These costs reflect the effort of creating a runtime environment for a parsed module.

The formula for instantiation cost per invocation is:

$$C_{\text{instantiaten},n} = k_{\text{instantiate}} \cdot Y_n$$

where:

- Y_n is the complexity of the n-th instantiation (e.g., memory usage, dependency resolution, or specific runtime operations).
- $k_{\text{instantiate}}$ is a scaling factor for instantiation costs.

Execution Costs: C_{execute}

Execution costs account for the runtime computation carried out during contract execution, including host function calls, arithmetic, state manipulation, and data access.

The formula for execution cost is:

$$C_{\text{execute}} = \sum_{i=1}^{I} \text{Cost}_i$$

where:

- I is the total number of instructions executed.
- Cost is the gas cost for the i-th instruction.

Execution Phases

Recording Mode

In Recording Mode, the Soroban host performs the following steps:

1. Identifies Wasm modules in the transaction's read footprint.
2. Parse each unique Wasm module and calculate parsing costs.
3. Store the parsed modules and their corresponding parsing costs in the ledger or a cache for reuse during future executions. This caching allows repeated invocations of the same module to avoid redundant parsing costs.

Enforcing Mode

In Enforcing Mode, the contract execution proceeds as follows:

1. Parse modules from the footprint and populate the cache.
2. Instantiate runtime instances (VMs) from the cached modules. This instantiation is done dynamically for each invocation.
3. Apply instantiation costs for each invocation based on runtime complexity, including memory usage, data handling, and other runtime operations.

Code Inspections

The Wasm parse module tracks parsing costs. Parsing occurs in recording mode and focuses on processing the Wasm bytecode and extracting the relevant metadata for cost calculation. This metadata typically includes:

- Number of functions in the module.
- Number of imports and exports.
- Size of data segments.

The parsing costs are calculated based on the Wasm module's size and complexity. They are then stored in the ledger or cache, which can be referenced during future executions.

The `ModuleCache` uses a `HashMap` to store parsed modules, allowing them to be reused in future transactions without re-parsing. The cache is initialized with a wasmi engine and configured using the host's budget. Modules are stored in a metered, ordered map to ensure efficient lookup and resource tracking. This caching system is essential for reducing redundant parsing operations and improving the efficiency of repeated contract invocations.

The cost meter tracks the computational costs associated with parsing. It ensures that the costs for parsing and instantiation are accurately calculated and applied at the proper times.

The `Parsedmodule` and their corresponding costs are serialized into XDR format and stored in the ledger. This allows the costs to be reused for future transactions involving the same module.

When a module is invoked, the VM is instantiated using the parsed module from the cache. During this instantiation process, the system applies the appropriate instantiation costs based on the complexity of the runtime environment created for the contract. Instantiation occurs in enforcing mode when contracts are invoked:

- Retrieves parsed modules from the cache.
- Crates a runtime instance (VM) to execute the contract.
- Dynamically applies instantiation costs for each invocation.

CAP-56 introduces a caching system that avoids redundant parsing within a single transaction, improving efficiency for repeated module invocations. CAP-54 relies on this system to:

- Cache parsed modules during recording mode, ensuring parsing costs are charged only once per unique module.
- Reuse cached modules during enforcing mode, dynamically applying instantiation costs for each invocation.

This integration between CAP-54 and CAP-56 ensures efficient module reuse and accurate cost metering across both transaction execution phases.

CAP-54’s parsing process leverages CAP-55's host function linking mechanism to minimize unnecessary imports.

The appendix of this document includes call graphs for functions that trigger parsing and instantiating modules: Appendix: Diagrams For Module Caching and Linking.

CAP-55: Only link host functions that are imported

This CAP proposed to change the availability of host functions to the Wasm interpreter. Prior to this change, all host functions were made available to guest code, regardless of whether they would be potentially used or not.

This change was implemented in commit 41b4ee3, which also contains the implementation of a module cache (CAP-56).

The code change was to, upon loading a module (into the module cache), go through its imports and identify the host functions that are imported into the module. Only these functions are then added to the Wasmi `Linker` data structure to make them available to the guest code.

In view of the *cost* of instantiating a module, which was made more granular with CAP-54, the module instantiation cost *decreases* because fewer functions have to be added to the linker (this is irrelevant unless the more granular cost model is used).

Code Inspection

Code for linking host functions resides in `soroban_env_host::vm::func_info` as well as in `soroban_env_host::vm::module_cache` and `soroban_env_host::vm`.
The code interacts tightly with the module caching mechanism (see CAP-56: Use a module cache per transaction) because the linker gets populated with host functions imported in *any* module involved in a transaction.

The host functions are linked when a new `VM` instance is created, either from a cached parsed module or from a new given module. The `ModuleCache` as a whole also entails a linker containing all host functions imported into any of its known modules. Both `ModuleCache::make_linker` and `ParsedModule::make_linker` will produce a resulting Wasmi `Linker` with all imported host functions.

A call graph of all functions involved in caching and loading modules can be found Appendix: Diagrams For Module Caching and Linking, illustrating further how and when the linker gets populated.

In `ModuleCache::make_linker`, using `ModuleCache::with_input_symbols` above it, the function `Host::make_linker` is called with all the host function symbols that were imported in any module of the cache (extracted within `ModuleCache::with_input_symbols` by iterating over `HOST_FUNCTIONS`) as its argument. `Host::make_linker` then iterates *again* over the `HOST_FUNCTIONS` and will add all host functions in its `BTreeSet` of symbols argument to the constructed linker.

This duplicates the iteration over the `HOST_FUNCTIONS` unnecessarily, but achieves the purpose of only adding functions that are in fact imported into a module in the module cache.
There is similar code in `parsed_module.rs` , `Host::make_linker` is called from `ParsedModule::make_linker`, too.
While `ParsedModule::with_imported_symbols` will indeed go over all imported symbols, the code in `ModuleCache`, which filters within the `ModuleCache::with_imported_symbols` method, will end up with the same import symbols because, as was confirmed by the client, the only legal imports are host functions.

CAP-56: Use a module cache per transaction

This CAP proposed to establish a "cache" of pre-loaded modules for an entire transaction, to avoid repeatedly parsing and instantiating Wasm modules. Transactions are typically accessing the same Wasm module/contract more than once to call different functions. The module cache prevents redundant work by reusing the same module when a new `VM` is constructed for each such call.

To implement the module cache, all required modules are collected during a "pre-flight" transaction run in *recording mode*. The collected modules are then pre-loaded into the host's `module_cache` when executing the transaction in *enforcing mode*.

The change was implemented in commit 41b4ee3 (together with CAP-55).

Code inspection

In *recording mode*, execution is provided with a *ledger snapshot* (see `invoke_host_function_in_recording_mode`). The module cache is not constructed until after the complete execution was recorded. Whenever a contract function in a module is called, the module is recorded as a read access in the `Footprint`. After execution finishes, the module cache will be "rebuilt" from the recorded storage, loading all modules present in the `Footprint` into the cache to charge their parsing cost. Instantiation cost has been charged during execution but the module parsing is not budgeted during execution when in recording mode.

In *enforcing mode* (see `invoke_host_function`), the execution starts from a given set of ledger entries (accessed items in storage), including stored modules to parse and initialise. Module parsing cost is charged when the module cache is set up, instantiation cost is charged during execution whenever a module is instantiated.

This distinction between the different storage modes makes the code harder to follow, but is a better choice than to maintain two different versions of the entire code for module instantiation and linking. Keeping the code for each storage mode adjacent makes it easier to compare and detect divergence in behaviour.

Independent of the storage mode, a new module could be uploaded within a transaction and then used in the same transaction. `vm.rs:Host::instantiate_vm` considers and mentions this

case. The new module will be loaded using its own separate `Engine` in this case.
Cost parameters for the loaded module are retrieved from storage together with the bytecode, and then threaded through a number of functions (`ParsedModule::new_with_cost_inputs`, `Vm::parse_module`, `ParsedModule::new_with_isolated_engine`, `ParsedModule::new`).

No design or implementation errors were found in the code that implements the module cache.

CAP-58: Support Contracts with Constructors

This CAP introduces support for contracts with a dedicated constructor function. The constructor function is named `__constructor`, using a prefix `__` which prohibits direct calls from Wasm code - only the *host* can call functions with this prefix.

As per description in the CAP, contract creation with a constructor is performed by a new host function `create_contract_with_constructor`, taking a (non-optional) vector of encoded arguments.

The constructor function can take arguments (arity and types must match and will be checked before executing the Wasm code), but may not return any result data. If the `__constructor` function returns any value other than `Val::Void`, the contract creation is considered as *failed*. Likewise, if the constructor code execution itself produces a failure, the contract creation fails. Contracts whose Wasm does *not* export a `__constructor` function can be created using the new host function, but then arguments *must not* be provided (i.e. the argument vector must be empty), or else the host function call likewise *fails* (this assumes a default `__constructor` function without arguments and without function body exists).
All failures lead to *reverting the entire transaction*, failures cannot be caught by the caller.

The semantics of the pre-existing host function `create_contract` (without constructor) is changed such that it will always call the function `__constructor` if it exists (and is externally visible). The `create_contract` variant will provide an empty argument vector, while `create_contract_with_constructor` will use the argument vector provided as a host function call argument.
The constructor call goes through `call_n_internal` via `call_contract_fn` and `VM::invoke_function_raw` to `VM::metered_function_call`, all the time carrying a boolean flag to indicate that a missing function should be treated as noop.

Authorization aspects of the new code are similar to authorizing the existing `create_contract` host function without constructor, but considers the constructor arguments as part of the authorization.

CAP-59: BLS Elliptic Curve Support

This CAP introduced host functions designed to enable cryptographic operations on the BLS12-381 elliptic curve within the Soroban smart contract platform. These functions provide a range of cryptographic primitives for advanced use cases, such as zero-knowledge proofs (ZKPs), multi-signature schemes, and verifiable credentials.

This change was implemented in commit 0497816, which introduced optimizations to the structure and resource metering of the host functions. The benefits of these functions include:

- Efficient cryptographic primitives for use cases like identity-based encryption, multi-party computation, and verifiable claims.
- Native execution of computationally expensive operations, such as elliptic curve scalar multiplication and pairing, reducing reliance on WASM for heavy cryptographic workloads.
- Improved developer usability with direct access to standard cryptographic operations in Soroban smart contracts.

Host Function Categories

CAP-59 defines host functions across four key categories, each addressing the cryptographic operations essential for elliptic curve cryptography on the BLS12-381 curve.

Field Arithmetic

Operations performed over the field $\mathbb{F}_p$ (the prime field) and $\mathbb{F}_{p^2}$ (the quadratic extension field), and $\mathbb{F}_r$ (the scalar field) are fundamental for elliptic curve operations, including:

- Addition: $(a + b) \mod r$.
- Subtraction: $(a - b) \mod r$.
- Multiplication: $(a \cdot b) \mod r$.
- Exponentiation: $(a^b) \mod r$
- Inversion: $a^{-1} \mod r$, where $a \neq 0$.

The corresponding host functions are: `bls12_381_fr_add`, `bls12_381_fr_sub`, `bls12_381_fr_mul`, `bls12_381_fr_pow`, and `bls12_381_fr_inv` for field arithmetic over $\mathbb{F}_r$. For $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$, those operations are implicitly covered by deserialization

(`fp_deserialize_from_bytesobj` and `fp2_deserialize_from_bytesobj`) and the elliptic curve related functions like `bls12_381_map_fp_to_g1` and `bls12_381_map_fp2_to_g2` .

These operations ensure that the elements of the field, which are used in elliptic curve computations, are valid and efficiently handled.

Elliptic Curve Arithmetic

The elliptic curve groups G_1 and G_2 are central to elliptic curve cryptography and pairing-based protocols. The key operations for these groups are:

- Point Addition: $P_{new} = P + Q$.
- Scalar Multiplication: $P_{new} = k \cdot P$, where k is a scalar.
- Multi-Scalar Multiplication: Computes $P_{new} = \sum_{i=1}^{n} k_i \cdot P_i$, efficiently combining multiple scalar multiplications.

The corresponding host functions are:

- `bls12_381_g1_add` and `bls12_381_g2_add` for point addition.
- `bls12_381_g1_mul` and `bls12_381_g2_mul` for scalar multiplication.
- `bls12_381_g1_msm` and `bls12_381_g2_msm` for multi-scalar multiplication.

These operations are used in applications such as signature aggregation, multi-signature schemes, and threshold signatures.

Hashing to Curve

The `bls12_381_hash_to_g1` and `bls12_381_hash_to_g2` functions map arbitrary data to points on the elliptic curve using the SWU (Specialized Weil Unitary) map. These functions ensure that arbitrary data (like messages or public keys) can be mapped to elliptic curve points efficiently and securely. The process includes:

- Validating the domain separator (`domain`).
- Hashing the message (`msg`) to a field element.
- Mapping the resulting field element to a point on the elliptic curve.

Additionally, the functions `bls12_381_map_fp_to_g1` and `bls12_381_map_fp2_to_g2` handle direct mappings from $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$, respectively, to the elliptic curve groups G_1 and G_2.

Pairing Operations

Bilinear pairing is computed between points in G_1 and G_2, essential for cryptographic protocols such as ZKPs and multi-signature schemes. The `bls12_381_multi_pairing_check` function computes pairings using the multi-miller loop followed by final exponentiation. The operation is defined as:

$$e(P, Q) = e(P \cdot Q)^{\textbf{exponent}}$$

where P is a point on G_1 and Q is a point on G_2.

The `bls12_381_multi_pairing_check` function checks that the input vectors for G_1 and G_2 are of the same length and non-empty and computes the final pairing result.

Execution Flow and Resource Metering

Recording Mode

The host functions are recorded during recording mode as part of the transaction footprint. This allows the system to:

- Track host functions usage for resource metering.
- Validate input parameters to ensure proper serialization and adherence to cryptographic constraints of BLS12-381.

Key actions during recording mode include:

- Validating curve points and field elements.
    - Curve points: Points on G_1 and G_2 must satisfy the elliptic curve equation.
    - Field elements: Ensure field elements are within the acceptable range.
- Tracking resource consumption to ensure proper metering for cost computation.

Enforcing Mode

During enforcing mode, host functions execute native cryptographic operations for efficient performance. Key steps include:

- Retrieving the transaction's relevant cryptographic data (field elements, elliptic curve points).
- Performing operations like scalar multiplication, point addition, or pairing.

- Results, such as elliptic curve points or scalar values, are serialized and returned to the smart contract for further processing.

Resource Metering

Each operation is metered based on complexity, ensuring high-cost operations like pairing and scalar multiplication are tracked to prevent Denial of Service (DoS) attacks. The total transaction cost is calculated as:

$$C_{\text{total}} = C_{\text{base}} + C_{\text{operation}}$$

where:

- C_{base} is the fixed invocation cost.
- $C_{\text{operation}}$ depends on the complexity of the operation, e.g., scalar multiplication or pairing.

Due to pairing's high complexity, fine-grained metering ensures computationally intensive operations do not lead to resource exhaustion or unintended service disruptions.

Code Inspection

Field Arithmetic

Functions like `bls12_381_fr_add`, `bls12_381_fr_sub`, `bls12_381_fr_mul`, `bls12_381_fr_pow`, and `bls12_381_fr_inv` implement basic operations over the scalar field. These functions are implemented using the Arkworks Rust library for BLS12-381, ensuring that field elements used in elliptic curve computations are valid and efficiently handled.

Elliptic Curve Arithmetic

Functions like `bls12_381_g1_add`, `bls12_381_g2_add`, `bls12_381_g1_mul`, and `bls12_381_g2_mul` handle point addition and scalar multiplication for both the G_1 and G_2 groups. These operations use Arkworks' optimized elliptic curve arithmetic methods for efficient point addition and scalar multiplication over the respective curve groups.

Hashing to Curve

The `bls12_381_hash_to_g1` and `bls12_381_hash_to_g2` functions map arbitrary data (`msg`) to points on the elliptic curve using the SWU map. The process involves checking that the domain separator (`domain`) is within valid length constraints (1 to 255 bytes), then applying the hashing function to the message (`msg`). The resulting hash is mapped to a point on the elliptic curve.

Pairing Operations

The pairing operation is a bilinear map between points in the G_1 and G_2 groups, using the `multi_miller_loop` followed by `final_exponentiation` to compute the pairing result. It ensures that the input vectors are of the same length and non-empty, implicitly assuming that all points are valid and lie on their respective curve.

Field Element Deserialization

The `fp_deserialize_from_bytesobj` function deserializes field elements of type `Fq` from byte objects. It ensures the serialized input data adheres to the expected format and size, enabling correct field element extraction for elliptic curve computations.

Output Serialization

Cryptographic results, such as elliptic curve points (e.g., `G1Affine` , `G2Affine`) or scalar values, are serialized into byte arrays using methods like `serialize_uncompressed_into_slice` and wrapped into objects such as `BytesObject` . This ensures compatibility with Soroban's input/output system.

CAP-60 - Using Register-Based Wasmi Engine

This CAP proposes updating the version of the Wasmi (Web Assembly Interpreter) from v0.31.0 to version v0.36.0.

Prior to this CAP the version of Wasmi supported was v0.31.0 which processes the instructions as a stack machine. CAP-60 changes the version of Wasmi to v0.36.0 which instead transforms the stack-based Wasm instructions to a register-based form for faster execution. Furthermore the change to v0.36.0 adds different modes of compilation, a choice between `Eager`, `Lazy`, and `LazyTranslation`. `Lazy` and `LazyTranslation` modes reduce computation cost by only translating required bytecode on-demand. The update to v0.36.0 also changes Wasmi internal fuel consumption metrics, meaning that Soroban budget analysis from v0.31.0 will not be accurate post upgrade, and will have to be re-analysed.

Translation and Validation

A Soroban smart contract is stored on the Stellar blockchain as mutable Wasm bytecode. When this bytecode is stored and loaded for execution, it is validated and translated by the wasmi interpreter. Translation is the conversion from Wasm instructions to Wasmi instructions. Validation is a well-formedness check performed on the Wasm (note NOT Wasmi) that must succeed for the bytecode to be considered valid. Validation is achieved in Wasmi through the wasmparser-nostd crate. Both translation and validation are metered functionalities in Wasmi.

Eager and Lazy

There are three modes of translation and validation possible in Wasmi v0.36.0:

`Eager` - Eager Translation and Eager Validation

`Eager` translation is equivalent to the translation that occurred prior to CAP-60 implementation, where loading the bytecode into the interpreter will initiate a full translation and validation of the bytecode. It is not always necessary for the bytecode to be translated and validated eagerly, and since they are metered this would cause unnecessary expenditure to the caller. `Eager` translation and validation is required when the bytecode is initially parsed prior to being uploaded to the ledger to ensure the stored bytecode is valid (see ParsedModule::new_with_isolated_engine).

`Lazy` - Lazy Translation and Lazy Validation

`Lazy` translation means the Wasmi interpreter will only translate and validate Wasm functions when they are about to be executed. This offers a large performance increase as there is only translation and validation of what is *necessary* for execution. The tradeoff is that there is no guarantee that the entire module is valid from the Wasmi interpreter's perspective, as there could be other functions that are not accessed that are invalid.

Soroban uses `Lazy` translation when creating a new `ModuleCache` (see ModuleCache::new), which can be initiated as part of 3 `Host` operations: calling a contract function (`Host::call_contract_fn`), getting a contract's protocol version (`Host::get_contract_protocol_version`), and during preflight footprint construction (`Host::pop_context`) (see Appendix: Diagrams For Module Caching and Linking -- Constructing the Module Cache diagram). Since the Wasm that is stored on the ledger is eagerly translated and validated prior to storage, there is no concern that the subsequent `Lazy` translation and validation Soroban performs is accessing valid functions inside an invalid module.

`LazyTranslation` - Lazy Translation and Eager Validation

The Wasm bytecode is eagerly validated, but translation is done lazy as required by called functions. This is a middle ground between the two previous options, it is not currently used in `rs-soroban-env` .

Wasmi Config

Wasmi can conditionally support Wasm proposals, and the configuration that Soroban uses is the same for all translation with Wasmi. All proposals are turned off except for:

- Bulk Memory Operations
- Mutable Global
- Sign Extension Operations

Wasmi Version Options

Upgrading Wasmi past v0.32.0 could mean choosing one of many targets, each with some advantages and disadvantages.

v0.36.5

This version of Wasmi is the most current version that has been audited with fuzzing and code review (See Wasmi - WebAssembly (Wasm) Interpreter). However this version has not addressed all the issues in the recommendations from the audit. Specifically, post-conditions to ensure soundness of the translation have not been implemented (still in active development).

v0.40.0 or main

Since v0.36.0, the latest version (v0.40.0) and main have decreased the total number of Wasmi instructions, and added some optimisations which improve macro-op fusion. These changes are certainly improvements from v0.36.0, however without the implementation of post-conditions and extra checks the soundness of the translation is not double-checked and errors might result in undefined behaviour. Since they have progressed past the audit, these guarantees are value to have implemented.

Future Versions

Versions v0.36.5 and greater offer improved performance, and thus improved cost, over the current v0.31.0. While this is desirable to merge soon, waiting for the postconditions and extra checks PRs to be completed should also be considered.

Metering Changes

Upgrading Wasmi to a version at or beyond v0.32.0 means the previous metering system will require recalculation as the fuel system in Wasmi has changed. Extra care should be taken when upgrading to ensure costs are accurate.

Storing Wasmi instead of Wasm

It should be considered that an improvement to the design between Wasmi and Soroban would be for serialised Wasmi to be stored in the ledger after `Eager` translation and validation. Then it would only be necessary to load the program, deserialise, and execute the program. This *may* be an improvement, but it is hard to determine how much performance benefit (if a benefit) would come from this approach. Deserialisation can be challenging, and it may require an engineering effort with little reward.

It should also be noted that storing the Wasm adds some flexibility with the supported Wasmi version being upgraded. Stored Wasmi may be valid for a particular version only, and thus this version needs to be supported permanently unless the stored bytecode was upgraded. The translation on demand each time should mean that versions can be upgraded without there being compatibility issues.

Informative Findings

The Informative Findings section highlights observations and recommendations that, while not classified as vulnerabilities, are worth addressing to improve code quality, maintainability, or performance. These findings may include suggestions for adhering to best practices, documentation improvements, or enhancements to the overall design.

Results of `cargo-audit` for Audited Repositories

Severity: Informative | Recommended Action: Fix Code | Not addressed by client

Rustsec tool `cargo-audit` was run on the two repositories to audit, and produced the following results:

- `rs-soroban-env` : No warnings
- `rs-stellar-xdr` : one warning RUSTSEC-2022-0078 related to a use-after-free in crate `bumpalo` , a transitive dependency of crate `serde_with` via an old version of the `chrono` crate and several other crates, including the `wasm-bindgen` family of crates.

  The dependency is only active when using optional features of `serde_with` . It could be avoided by an explicit optional dependency on `bumpalo` in a newer version.
  Upgrading all dependencies to latest available versions with `cargo update` also removes the warning.

Recommendations

Upgrade dependencies

Status

Reported to the client

Fuzzing Campaign Discussion and Findings

This section details the fuzzing campaign conducted during the audit, including the methodology used, tools employed, and test harnesses developed to stress-test the codebase. It highlights vulnerabilities and unexpected behaviors identified through dynamic analysis and discusses the significance of these findings in ensuring code robustness under edge-case scenarios.

Description of Fuzzing Approach

Fuzzing Libraries and Hardware

We used a fuzzing application and library called `honggfuzz`. We prefer this fuzzer because it performs very well, automatically using multiple threads and persistent processes to run test cases.

The fuzzing process was distributed across several machines with varying specifications:

- AMD Ryzen 9 7950X (16 core / 32 thread) with 128 GB RAM
- Intel Core i9-13900K (24 core / 32 thread) with 64 GB RAM
- Threadripper 1950X (16 core / 32 thread) with 64GB RAM

Fuzzing Targets

To identify fuzzing targets, we drew from multiple sources

There were a few pre-existing targets in the project either already in the form of fuzzing targets, or in the form of property tests that we converted to fuzz targets. We also needed to change these targets to use the honggfuzz libraries.

For areas lacking fuzzing coverage, we manually wrote new targets. We took inspiration from code walkthrough meetings with the client, existing unit tests throughout the project, and areas of interest shared by the client.

Overview of Fuzzing Targets and Run Times

expr

Under `soroban-env-host/fuzz/` there exists a fuzzing target named `expr.rs`. It uses `libfuzzer`, which we changed to use `honggfuzz` with minimal effort.

proptest_val_cmp

This is a property test in `rs-soroban-sdk` which tests the roundtrip conversion between `Val` and `ScVal`, checking the invariant that a comparison result in one form is equivalent to the result in the other form. This uses the `proptest` library which only ran 10000 iterations at a time. We replaced it with `honggfuzz` to fuzz over it continuously.

proptest_scval_cmp

This is similar to `proptest_val_cmp`, only the test begins with `ScVal`s instead of `Val`s. There exists an issue with this test in that many arbitrarily generated `ScVal`s aren't valid and cause the test case to end prematurely, reducing the yield of useful test cases in our fuzzing runs.
To address this issue, an `Arbitrary` generator for `ScVal` was manually implemented which would only generate valid values[1] (code was shared with the client). However, this generator causes issues of performance and memory consumption (because of the recursive nature of the ScVal variants `ScVec` and `ScMap`). The generator was nevertheless used in a modified target `scval_cmp`, which caused timeouts but remains a valid target.

scval_bytes_roundtrip

This was derived from a property test for the default `ScVal` generator, which was measuring how many valid values were generated.
In each iteration, an `ScVal` was generated and then serialised to a byte array. Where successful (i.e. the `ScVal` was valid), a subsequent deserialisation from a byte array into a host value was performed, followed by another serialisation into a second byte array. The resulting byte arrays are expected to be identical.
Using the modified generator described above, no invalid values should be observed, except for

cases where the host's XDR limits would be exceeded (cases which are also likely to cause timeout or memory exhaustion in the generator).

Mock setup for testing linear memory operations

In order to test host functions that perform linear memory operations on the guest code's `memory`, the host functions will access the `Vm` of the guest code, and thus require the `Vm` and a `VmCaller` data structure to be available and valid. It would be possible to generate Wasm modules that call host functions with random arguments. However, an alternative setup can be to instead emulate the environment of such host function calls in a mocked-up setting with a `Vm` and `VmCaller` that was manually created.

We explored this second alternative, and were able to develop a suitable `Vm` environment for calling linear-memory host functions with minimal changes to the visibility of some host functions (notably exposing context `Frame` s and host object creation outside the crate). Code was shared with the client as a patch file with the fuzzing targets as well as necessary changes to host code.

fuzz_linear_memory

A first fuzzing target for linear memory operations was developed which writes, modifies, and reads back, random data as a host `String` and as a `BytesObject`. This target was not run for very long because its coverage plateaued early (using random *data* does not lead to too many different code paths), even when randomising the budget available to the host in order to produce out-of-budget failures during the write/read operations.

read_corrupted and read_corrupted2

The second fuzzing target we developed constructs a host `Map` in the host from an arbitrary keys vector, and then writes the `Map` s keys and values to linear memory using host functions. Subsequently, the written data is modified in a random location, before an attempt to construct a fresh `Map` from the linear memory contents. This attempt is expected to fail in many cases because the keys or values got corrupted arbitrarily. Importantly, none of the failures should be an `InternalError` nor an outright `panic` on the host.

This target quickly uncovered a host failure, which was then found to be the same failure as one discovered by the `expr` target earlier. The host code was patched to avoid this failure before running the target(s) for longer.

The first version of this target used `ValidScVal` to generate the arbitrary key vector. This unfortunately led to many timeouts, and exhausted memory when run with larger timeouts, due to the shortcomings of the `ValidScVal` generation described before. A second target `read_corrupted2` was developed which uses the default (derived) `ScVal` generator but replaces all invalid values by `VOID`. This target displayed a much higher throughput, at the price of reducing the randomness of the generated `Map` in use for each iteration (we expect many `Map` s to be identical singleton maps `VOID -> VOID`).

Budget Exhaustion during Error Handling (error_budget)

The error handling in the host is a good target for fuzzing. Many host functions have error cases which return early, and it isn't clear how budget exhaustion may affect the execution beyond that point. It could be that if the budget runs out during error handling, some cleanup that needs to happen gets skipped. Assuming a case like this does occur, identifying it can be tricky because the behavior that manifests from the corrupted state is difficult to predict. Regardless, we began working on a target to hit as many of these error cases as possible and vary the budget, in the hopes of hitting a case where some state corruption triggers a panic or an InternalError.

Run times

target	run time	issues found
expr	22.5 days	1
val_cmp	13 days	0
scval_cmp (original)	1 day	0
scval_cmp (modified generator)	4.5 days	0
scval_bytes_roundtrip	4.75 days	0
fuzz_linear_memory	1.5 days	0
read_corrupted	6.75 days	1 (same as expr)
read_corrupted2	15 days	0
error_budget	4 days	0

1. To ensure that generated `ScVal` ues are valid, two invariants not implied by types must be established: **1)** For `Symbol` values, characters are limited to `[_a-zA-Z0-9]` (instead of arbitrary strings), and **2)** `Map` values are represented as a vector of `(key, value)` pairs sorted by `key` s and without duplicate keys. This second invariant is expensive to establish, given that each `key` and `value` can recursively again be a `Map` value. ↩

[F1] Panic discovered in `expr` fuzzing target

Not addressed by client

Context

An input to the `expr` target revealed a call to `SmallSymbol::try_from_bytes` with a byte array of `{0,0}`. These zeroed bytes are invalid inputs and causes the host to throw an internal error.

Severity

This was deemed as low severity, as the error is caught by the host and terminates the transaction.

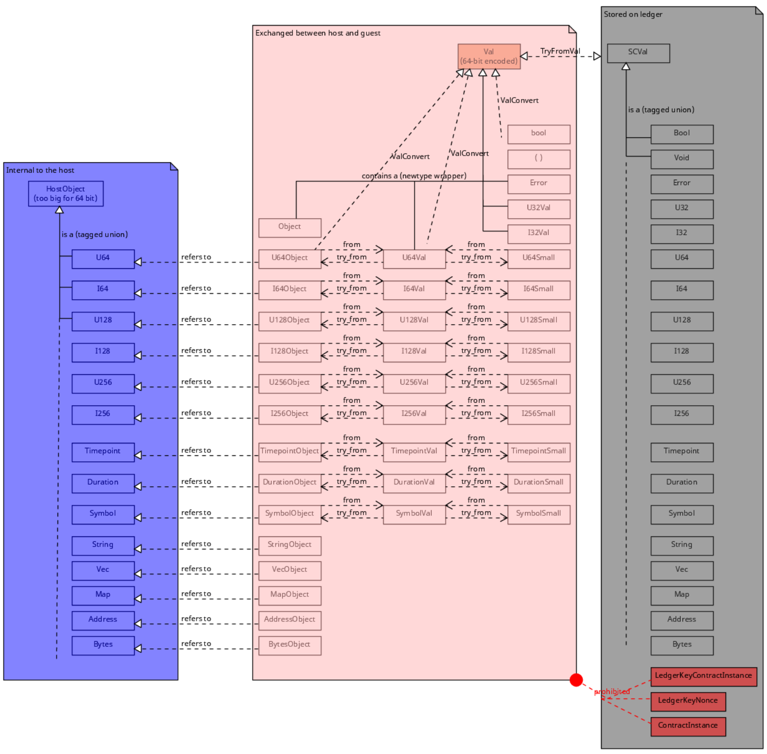
Appendix

The Appendix provides supplementary material relevant to the audit, such as detailed explanations, diagrams, or technical artifacts that support the findings and methodology described in earlier sections. This section serves as a resource for developers and stakeholders seeking additional context or deeper insights into the audit process.

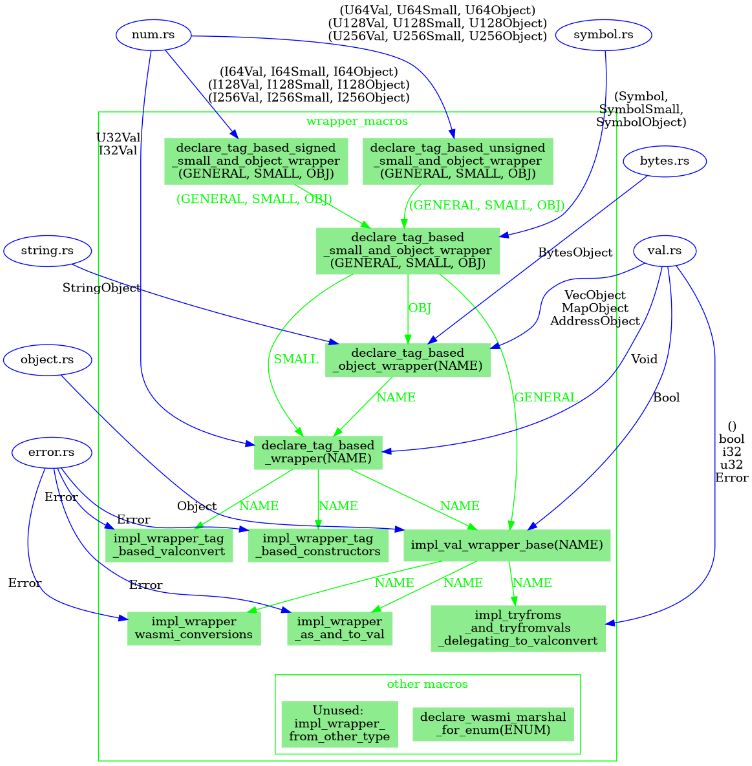
Appendix: Data Type Macros and Diagram for `rs-soroban-env`

The host's `Val` type is a tagged sum type representing different data (different integral number types, boolean values, symbols, strings, addresses, vectors, maps).
The last 8 bits are used as a tag, while the payload data is either directly included (if it fits into 56 bits), or else a reference to a *host object* which gets created in the host using host functions or when using ledger data converted from an XDR representation.

Special constructors for each of the different data types exist and wrap the internal representation for uses where a certain specific type is expected.
Some of the represented types therefore have both a "small" and an "object" representation, as subtypes of the general variant in question.



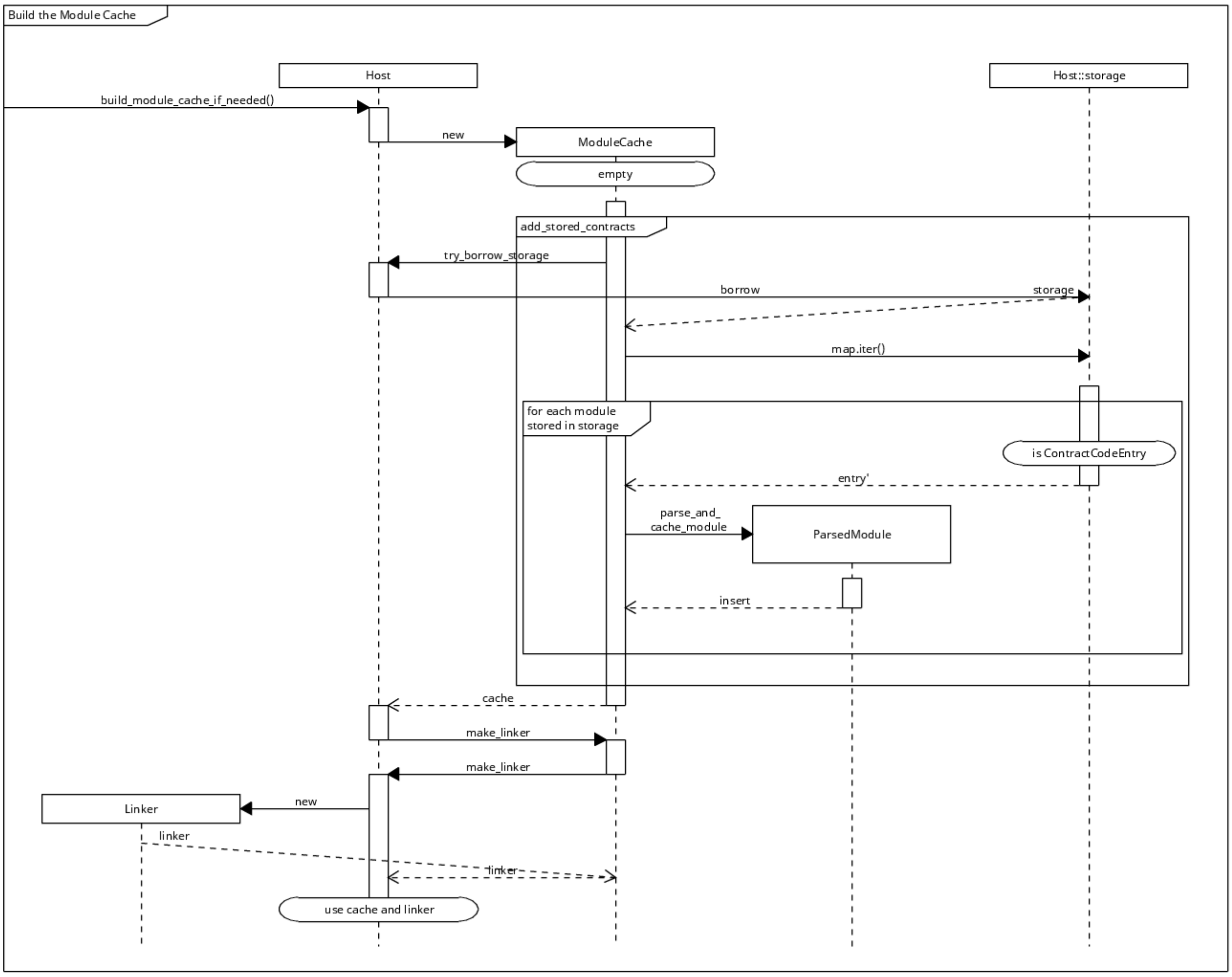
A number of macros defined in `soroban-env-common::wrapper_macros` are used extensively to define all important data types for interaction between host and guest code, as well as conversion to the internal types from ledger data (XDR). These macros are called from different modules within `soroban_env_common` to define these more specific data types around the `Val` type, and used throughout by the host and guest code to exchange data. The following diagram illustrates the usage of the macros (in green) to define the types in question (in blue).



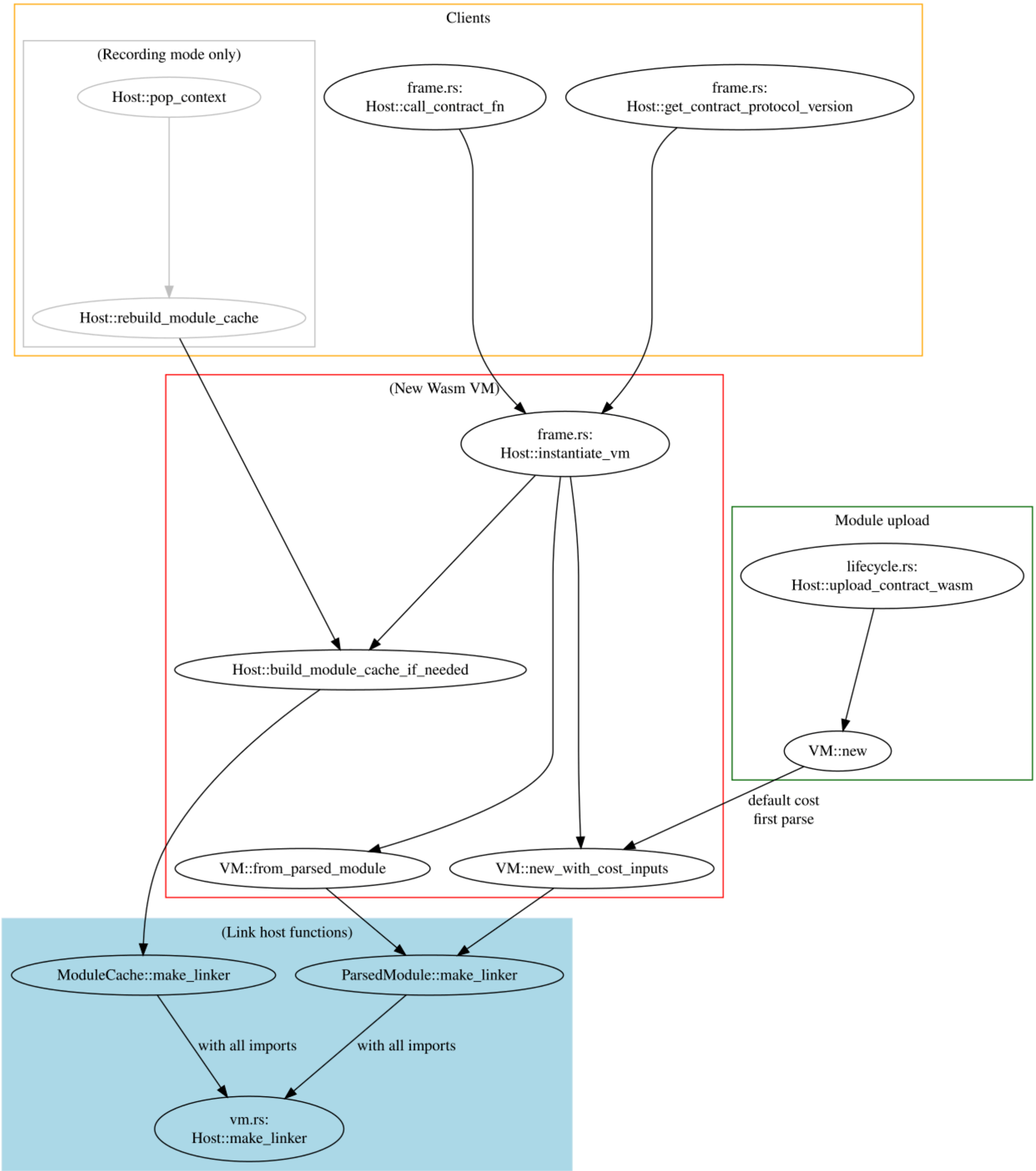
Macro name in `wrapper_macros.rs`	Purpose and callees
`impl_wrapper_tag_based_valconvert`	`ValConvert` instance
`impl_wrapper_tag_based_constructors`	`from_body` and `from_major_minor`
`impl_tryfroms_and_tryfromvals_delegating _to_valconvert`	`TryFrom<Val>` and `TryFromVal<E, Val>` instances
`impl_wrapper_wasmi_conversions`	`WasmiMarshal` instance (`Wrapper -> wasmi::Val`)
`impl_wrapper_as_and_to_val`	`Wrapper <---> Val` and `as_mut/as_ref`
`impl_wrapper_from_other_type`	`Other --> Wrapper` , `Wrapper --> Result<Other, Err>` **UNUSED**
`impl_val_wrapper_base`	calls `as_and_to_val` , `wasmi_conversions` , `tryfroms`
`declare_wasmi_marshal_for_enum`	`WasmiMarshal` instance for enum (maps to `I64`)
`declare_tag_based_wrapper`	wraps `Val` with data in a specific struct, and calls `wrapper_base` , `tag_based_valconvert` , `tag_based_constructors`
`declare_tag_based_object_wrapper`	wraps `Val` with `Object` ref in a specific struct, `Compare` instance for specific objects (compared as objects). Calls `tag_based_wrapper`
`declare_tag_based_small_and_object_wrappers`	declares both wrappers for small and object-based `Val` s, and a general wrapper, comparisons/equality for small, and conversions between general and specific types. Calls `tag_based_wrapper` , `tag_based_object_wrapper`
`declare_tag_based_unsigned_small_and_object_wrappers`	above, and `Ord` instance based on `Val::get_body` . Calls `tag_based_small_and_object_wrapper`
`declare_tag_based_signed_small_and_object_wrappers`	above, and `Ord` instance based on `Val::get_signed_body` . Calls `tag_based_small_and_object_wrapper`

Appendix: Diagrams For Module Caching and Linking

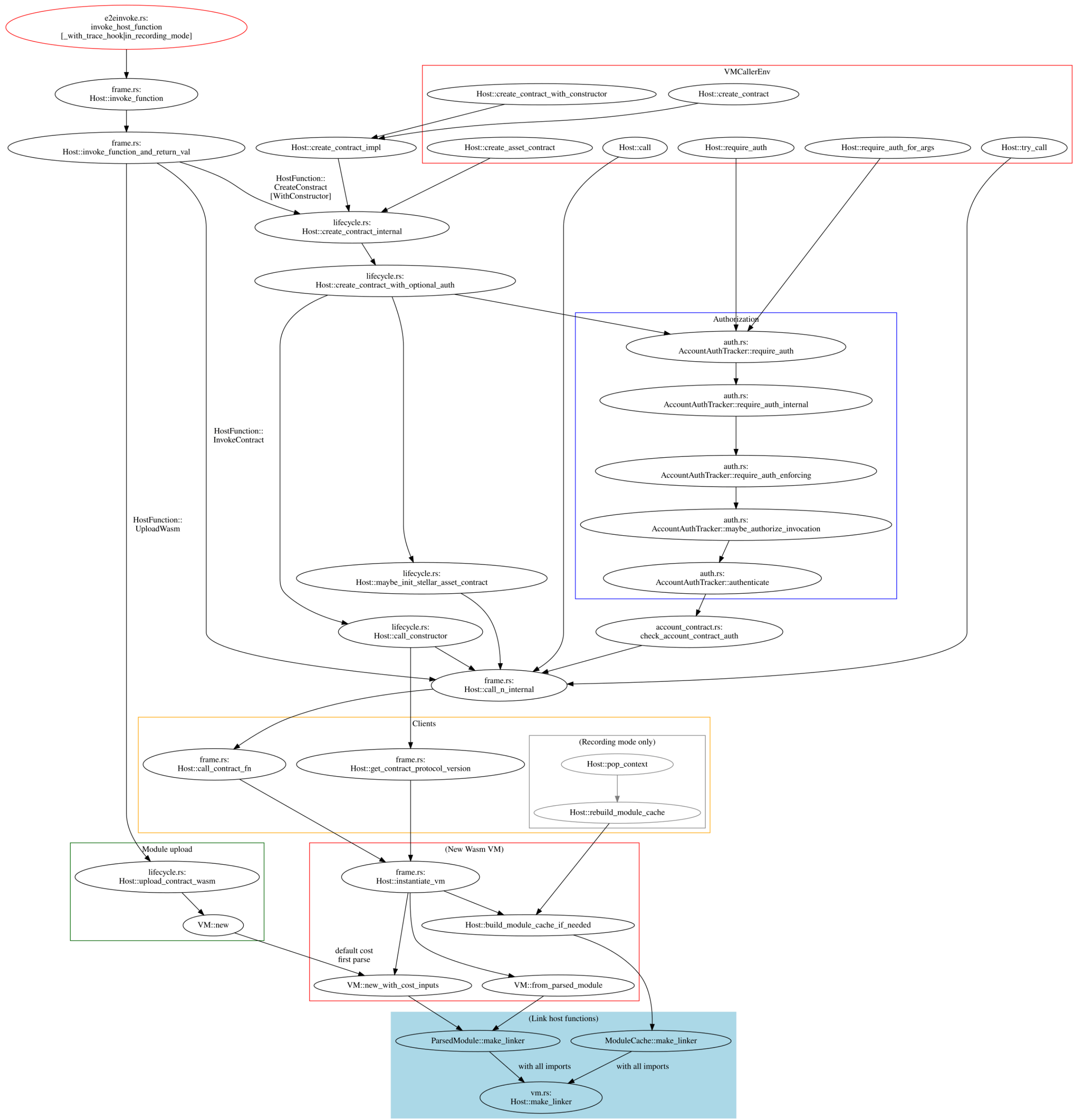
Sequence diagram: Module loading



Constructing the Module Cache



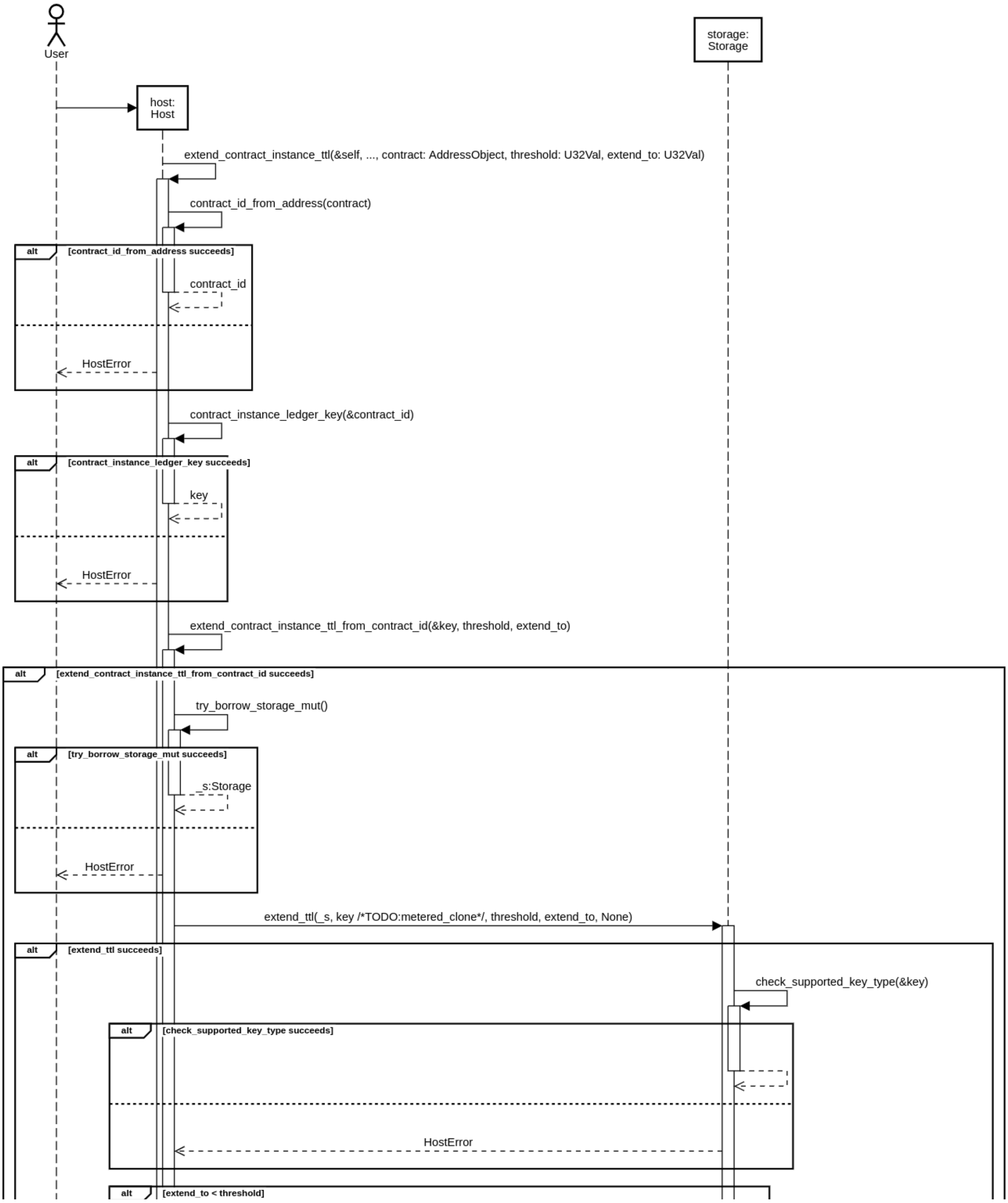
Calling Host Functions (triggers module cache rebuild)

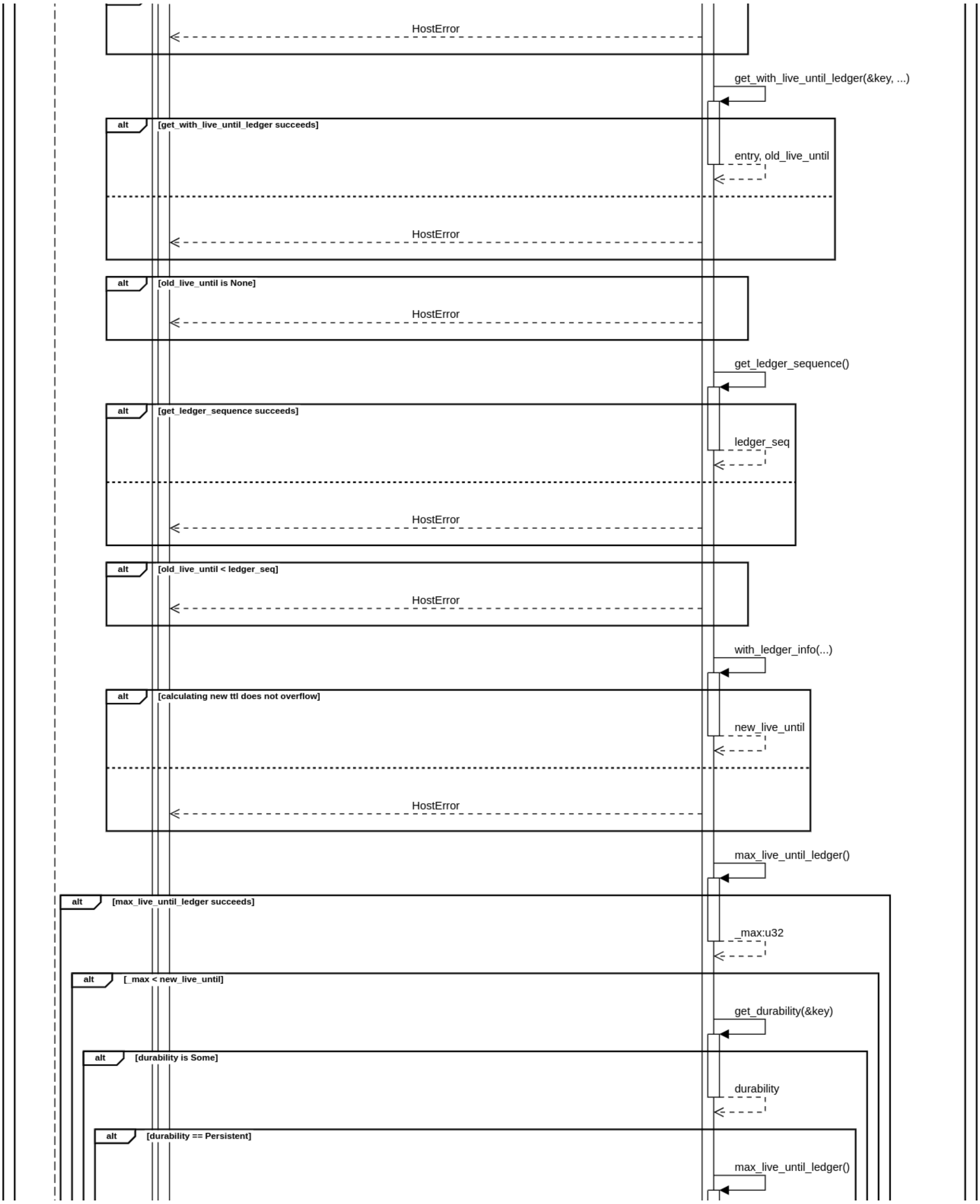


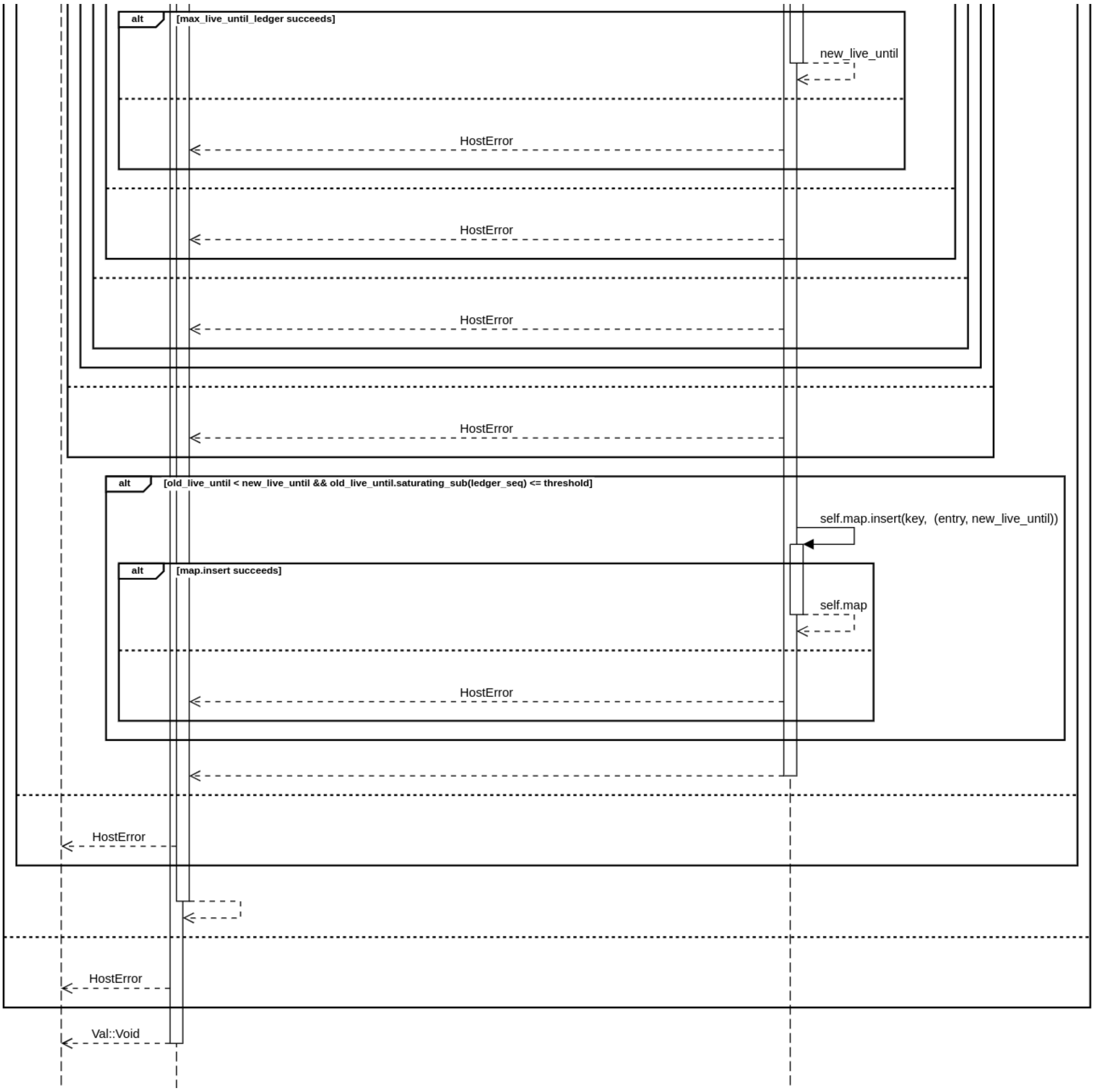
e2einvoke.rs:
invoke_host_function
[_with_trace_hook|in_recording_mode]
frame.rs:
Host::invoke_function
frame.rs:
Host::invoke_function_and_return_val
VMCallerEnv
Host::create_contract_with_constructor
Host::create_contract
Host::create_contract_impl
Host::create_asset_contract
Host::call
Host::require_auth
Host::require_auth_for_args
Host::try_call
HostFunction::
CreateConstract
[WithConstructor]
lifecycle.rs:
Host::create_contract_internal
lifecycle.rs:
Host::create_contract_with_optional_auth
Authorization
auth.rs:
AccountAuthTracker::require_auth
auth.rs:
AccountAuthTracker::require_auth_internal
auth.rs:
AccountAuthTracker::require_auth_enforcing
auth.rs:
AccountAuthTracker::maybe_authorize_invocation
auth.rs:
AccountAuthTracker::authenticate
HostFunction::
InvokeContract
HostFunction::
UploadWasm
lifecycle.rs:
Host::maybe_init_stellar_asset_contract
lifecycle.rs:
Host::call_constructor
account_contract.rs:
check_account_contract_auth
frame.rs:
Host::call_n_internal
Clients
frame.rs:
Host::call_contract_fn
frame.rs:
Host::get_contract_protocol_version
(Recording mode only)
Host::pop_context
Host::rebuild_module_cache
Module upload
lifecycle.rs:
Host::upload_contract_wasm
VM::new
(New Wasm VM)
frame.rs:
Host::instantiate_vm
Host::build_module_cache_if_needed
default cost
first parse
VM::new_with_cost_inputs
VM::from_parsed_module
(Link host functions)
ParsedModule::make_linker
ModuleCache::make_linker
with all imports
with all imports
vm.rs:
Host::make_linker

Appendix: `extend_ttl` Sequence Diagram

Extension of TTL for Ledger Entries




HostError
get_with_live_until_ledger(&key, ...)
alt
[get_with_live_until_ledger succeeds]
entry, old_live_until
HostError
alt
[old_live_until is None]
HostError
get_ledger_sequence()
alt
[get_ledger_sequence succeeds]
ledger_seq
HostError
alt
[old_live_until < ledger_seq]
HostError
with_ledger_info(...)
alt
[calculating new ttl does not overflow]
new_live_until
HostError
max_live_until_ledger()
alt
[max_live_until_ledger succeeds]
_max:u32
alt
[_max < new_live_until]
get_durability(&key)
alt
[durability is Some]
durability
alt
[durability == Persistent]
max_live_until_ledger()



Appendix: `unsafe` Rust code

Uses of `unsafe` Rust code in `rs-soroban-env` (crates `soroban-env-[common|guest|host]`)

The crate `rs-soroban-env` contains a number of instances of unsafe Rust code.
All code that was inspected was marked `unsafe` either to *declare* an unsafe function (`DECL` in

the table), or to *call* an unsafe function (`CALL` below). More often than not, the unsafe function being called was one of the functions declared unsafe in other parts of the same crate.

All `unsafe` code relates to data type conversions between the 64-bit representation of values and their internal representation in the host code (`Val`).
The intention of these `unsafe` attributes on functions was clarified in a dialog with the client. The `unsafe` attribute is used to point out that these functions either make internal, potentially unstable, representations observable (e.g., `SymbolSmall::get_body` provides the internal representation with 6 bits per character), or do not ensure certain invariants (e.g., `from_body_and_tag` does not guarantee that tag and bit pattern in the body of a `Val` correspond).
The code does not usually contain dedicated safety comments for the declared functions.

In `soroban-env-common`

File : Line	Type: Comment
soroban-env-common/src/compare.rs : 120/121	CALL `unchecked_from_val` to yield comparison items, tags checked before (macro call sites)
soroban-env-common/src/convert.rs : 393	CALL `Error::unchecked_from_val` , tag checked before
soroban-env-common/src/convert.rs : 431	CALL `SymbolSmall::unchecked_from_val` , tag checked before
soroban-env-common/src/convert.rs : 501	CALL `from_body_and_tag` , size checked beforehand
soroban-env-common/src/convert.rs : 505	CALL `from_body_and_tag` , size checked beforehand
soroban-env-common/src/convert.rs : 509	CALL `from_body_and_tag` , size checked beforehand
soroban-env-common/src/convert.rs : 513	CALL `from_body_and_tag` , size checked beforehand
soroban-env-common/src/convert.rs : 518	CALL `from_body_and_tag` , size checked beforehand
soroban-env-common/src/convert.rs : 523	CALL `from_body_and_tag` , size checked beforehand
soroban-env-common/src/convert.rs : 527	CALL `from_body_and_tag` , size checked beforehand
soroban-env-common/src/convert.rs : 531	CALL `from_body_and_tag` , size checked beforehand
soroban-env-common/src/error.rs : 312	CALL `from_major_minor` , contract error (error code unrestricted)
soroban-env-common/src/error.rs : 317	CALL `from_major_minor` , types ensuring correct data
soroban-env-common/src/num.rs : 18	CALL `from_major_minor_and_tag` to convert `u32`
soroban-env-common/src/num.rs : 23	CALL `from_major_minor_and_tag` to convert `i32`
soroban-env-common/src/num.rs : 131	CALL `from_body_and_tag` in `try_from` , checked beforehand
soroban-env-common/src/num.rs : 144	CALL `from_body_and_tag` in `try_from` , checked beforehand
soroban-env-common/src/num.rs : 157	CALL `from_body_and_tag` in `try_from` , checked beforehand
soroban-env-common/src/num.rs : 170	CALL `from_body_and_tag` in `try_from` , checked beforehand
soroban-env-common/src/num.rs : 211	CALL `from_body_and_tag` in `try_from` , checked beforehand
soroban-env-common/src/num.rs : 224	CALL `from_body_and_tag` in `try_from` , checked beforehand
soroban-env-common/src/num.rs : 243	CALL `from_body_and_tag` in `try_from` , checked beforehand
soroban-env-common/src/num.rs : 262	CALL `from_body_and_tag` in `try_from` , checked beforehand
soroban-env-common/src/num.rs : 319	CALL `from_body` , infallible case (32-bit size)
soroban-env-common/src/num.rs : 332	CALL `from_body` , infallible case (32-bit size)
soroban-env-common/src/num.rs : 345	CALL `from_body` , infallible case (32-bit size)
soroban-env-common/src/num.rs : 358	CALL `from_body` , infallible case (32-bit size)
soroban-env-common/src/num.rs : 371	CALL `from_body` , infallible case (32-bit size)
soroban-env-common/src/num.rs : 384	CALL `from_body` , infallible case (32-bit size)
soroban-env-common/src/object.rs : 20	DECL `unchecked_from_val` in `impl ValConvert`
soroban-env-common/src/object.rs : 34	CALL `from_major_minor_and_tag` (handle and tag not checked)
soroban-env-common/src/object.rs : 98	DECL `ValConvert::unchecked_from_val`
soroban-env-common/src/symbol.rs : 140-145	CALL `unchecked_from_val` to implement `Symbol` comparison
soroban-env-common/src/symbol.rs : 163	CALL `SymbolSmall::from_body` , checked beforehand
soroban-env-common/src/symbol.rs : 255	CALL `SymbolSmall::from_body` with constructed well-formed payload
soroban-env-common/src/symbol.rs : 263	DECL `SymbolSmall::get_body` (rationale: exposes internal representation)
soroban-env-common/src/symbol.rs : 330	CALL `str::from_utf8_unchecked` for `SymbolStr::as_ref` . Relies on sound `SymbolStr` construction by `EnvBase`
soroban-env-common/src/symbol.rs : 353	CALL `SymbolObject::unchecked_from_val` (relies on `Symbol` construction)

File : Line	Type: Comment
soroban-env-common/src/symbol.rs : 449	CALL `SymbolSmall::from_body` on constructed payload
soroban-env-common/src/val.rs : 197	CALL `transmute` on `Tag` , checked beforehand
soroban-env-common/src/val.rs : 315	DECL `Bool::unchecked_from_val` in `impl ValConvert`
soroban-env-common/src/val.rs : 381	DECL unsafe function `unchecked_from_val` in `ValConvert` trait
soroban-env-common/src/val.rs : 392	CALL `unchecked_from_val` , checked beforehand
soroban-env-common/src/val.rs : 470	DECL `()::unchecked_from_val` in `impl ValConvert`
soroban-env-common/src/val.rs : 479	DECL `bool::unchecked_from_val` in `impl ValConvert`
soroban-env-common/src/val.rs : 500	DECL `u32::unchecked_from_val` in `impl ValConvert`
soroban-env-common/src/val.rs : 511	DECL `i32::unchecked_from_val` in `impl ValConvert`
soroban-env-common/src/val.rs : 658	CALL `Error::unchecked_from_val` , result checked by `try_from`
soroban-env-common/src/val.rs : 734	DECL `unsafe` function `Val::from_body_and_tag`
soroban-env-common/src/val.rs : 740	DECL `unsafe` function `Val::from_major_minor_and_tag`
soroban-env-common/src/val.rs : 773	CALL `from_body_and_tag` to encode `Void`
soroban-env-common/src/val.rs : 779	CALL `from_body_and_tag` to encode `Bool`
soroban-env-common/src/val.rs : 826	CALL `unchecked_from_val` for printing `Error` s
soroban-env-common/src/val.rs : 839	CALL `unchecked_from_val` for printing `SymbolSmall`
soroban-env-common/src/wrapper_macros.rs : 14	DECL `unchecked_from_val` in `impl ValConvert` in macro
soroban-env-common/src/wrapper_macros.rs : 28	DECL `from_body` in `wrapper_tag_based` macro
soroban-env-common/src/wrapper_macros.rs : 34	DECL `from_major_minor` in `wrapper_tag_based` macro
soroban-env-common/src/wrapper_macros.rs : 90	CALL `unchecked_from_val` in `wasmi_conversions` macro
soroban-env-common/src/wrapper_macros.rs : 246	DECL `from_handle` in `declare_tag_based` macro
soroban-env-common/src/wrapper_macros.rs : 281	DECL `unchecked_from_val` in `impl ValConvert` in macro

In `soroban-env-guest`

File : Line	Code
soroban-env-guest/src/guest.rs : 211	DECL All host functions are declared `extern "C"` so they are called `unsafe` ly within `impl Env for Guest`

In `soroban-env-host`

File : Line	Code
soroban-env-host/src/host/conversion.rs : 360	CALL `ScValObject::unchecked_from_val` . Unnecessarily large `unsafe` block
soroban-env-host/src/host_object.rs : 166	CALL unsafe `$TAG::from_handle` (macro) call (handle not known to refer to correct object type). Actually `$TAG::new_from_handle` is just as unsafe but not marked as such. Used once, where it is safe.

In Test code

File : Line	Code
soroban-env-common/src/symbol.rs : 597	CALL `get_body` in test code
soroban-env-host/src/test/dispatch.rs : 113	CALL `U64Object::from_handle` for test
soroban-env-host/src/test/dispatch.rs : 118	CALL `I64Object::from_handle` for test
soroban-env-host/src/test/dispatch.rs : 123	CALL `TimepointObject::from_handle` for test
soroban-env-host/src/test/dispatch.rs : 128	CALL `DurationObject::from_handle` for test
soroban-env-host/src/test/dispatch.rs : 133	CALL `U128Object::from_handle` for test
soroban-env-host/src/test/dispatch.rs : 138	CALL `I128Object::from_handle` for test
soroban-env-host/src/test/dispatch.rs : 143	CALL `U256Object::from_handle` for test
soroban-env-host/src/test/dispatch.rs : 148	CALL `I256Object::from_handle` for test
soroban-env-host/src/test/dispatch.rs : 153	CALL `BytesObject::from_handle` for test
soroban-env-host/src/test/dispatch.rs : 158	CALL `StringObject::from_handle` for test
soroban-env-host/src/test/dispatch.rs : 163	CALL `SymbolObject::from_handle` for test
soroban-env-host/src/test/dispatch.rs : 168	CALL `VecObject::from_handle` for test
soroban-env-host/src/test/dispatch.rs : 173	CALL `MapObject::from_handle` for test
soroban-env-host/src/test/dispatch.rs : 178	CALL `AddressObject::from_handle` for test
soroban-env-host/src/test/event.rs : 292	CALL `from_handle` with forged handle for test
soroban-env-host/src/test/hostile.rs : 148	CALL `from_handle` with forged relative handle `0xffff0` for test
soroban-env-host/src/test/hostile.rs : 493	DECL local variant of `Val::from_body_and_tag` for test
soroban-env-host/src/test/symbol.rs : 186	CALL `core::str::from_utf8_unchecked` with test data

Appendix: Entity-Relationship Diagram for `AuthManager`