Fuel OtterSec audit recap

Suggestions

Stack and heap

It would it make sense to introduce some sort of address segment separation so that the same address can't be used for both memory regions

Multiple Merkle Tree implementation

The current situation with multiple `MT` implementations is not ideal. SumTree seems redundant

Removal of Bug in favour of unreachable

There's technically no reason for `Bug` to exist. I'd like to replaced it with `unreachable!` everywhere.

Binary merkle trees (fuel-merkle)

Binary merkle trees uses in-order node ids for internal bookkeeping. The max in-order id could be up to twice the leaf count, but both are stored in `u64`. This means that if a tree contains `> 2^63` nodes, the in-order node id will overflow. Practically, it should be impossible to have a tree of such size, but for additional safety, we still recommend either adding checks to catch overflows, or use u128 for in-order ids instead.

Binary merkle trees (fuel-merkle)

Could this condition be simplified to just `num_leaves - num_leaves_left_subtree <= 1`? Our understanding is that this check is only necessary to prevent underflow here. And since `num_leaves_left_subtree` is already handled above, the additional check seems redundant
Other suggestions in the same thread

A. Use checked arithmetic in all areas - https://github.com/FuelLabs/fuel-vm/issues/170
B. Remove misleading comment about placeholders inside `Node::create_node_on_path()`
C. Simplify check condition during BMT proof verification

Mark test only structures as such (fuel-vm)

Would it be possible to mark test only structures as `test_only`? Such as MemoryStorage

Several suggestions (fuel-vm, fuel-tx)

1) The entire `fuel-tx/src/const.rs` file seems to contain incorrect offsets, and the values are not used anywhere. Deprecated legacy code should be removed.
2) There is a redundant arm in the match here that can be removed.
3) We're wondering whether it makes sense to use `gas_costs.s256().resolve((bytes + ChainId::SIZE) as u64)` in those places (1), (2), (3) since calculation of `tx_id` prepends `chain_id` to the serialized tx.
4) We noticed several presumably inaccurate implementations in `fuel-vm/src/storage/memory.rs`. But since the team mentioned `MemoryStorage` is for test only, it probably doesn't matter. But still including those here for reference
    a) It is generally inappropriate to use `hash(block_height)` as `block_hash` here
    b) Block time are generally not constant as assumed here
    c) This fails to include the max possible key ('\xff\xff...') in the iterator and may lead to incorrect results
    d) Iterating all keys may result in incorrect `all_set_key` when there are other keys outside of the removal range here. The iteration may also appear to be more costly than necessary.
5) Based on (4), it would be immensely helpful if the team could mark all test-only functions as `#[test]`. We have reviewed several functions, only to realize that it is only used in tests later. Proper anotations would make the code easier to read as well as help us focus on the real code that requires audit.
6) Regarding structure serialization, we are wondering whether there are better ways to keep track of field offsets (e.g. input). It seems like using a `macro` for automatic offset derivation would make the codebase cleaner as well as more robust against coding mistakes.
7) We would like to suggest hardening heap memory checks. The exact changes are this to `(self.hp..=heap_end).contains(&range.end)` + this to `(self.hp..heap_end).contains(&range.start)`.

Other Fuelvm suggestions

1. Is this code redundant since `post_execute` here does the same thing.
2. This check seems redundant since `write_bytes` also performs the same check.
3. The spec does not properly reflect implementation of `gtf` instruction. `GTF_OUTPUT_CONTRACT_BALANCE_ROOT` / `GTF_OUTPUT_CONTRACT_STATE_ROOT` are included in the spec, but not implemented. `InputCoinTxPointer` / `InputContractTxId` / `InputContractOutputIndex` are implemented, but not documented, and `ScriptReceiptsRoot` is included in the GTF enum, but not documented, and not handled within `get_transaction_field`.
4. We want to confirm whether `ecal` is intended to return an error in the default implementation, since the comment and code doesn't match up here.
5. Is the `GasUnit` used anywhere? This seems unused throughout the codebase.

Sway lib suggestions

1. We are wondering whether it makes sense to add checks to functions that fetch value from `fp` to prevent misuse outside of contracts. (1) / (2) / (3) / (4)
2. Is it intended for `StorageSlice` of `len==0` to return `None` on `read_slice`?
3. Should we release the storage used on pops? This might affect the total storage usage for contracts that never calls clear on vecs, but may grow a vec then remove items within it. (1) / (2) / (3) / (4)
4. Is this trait deprecated?
5. We noticed that a lot of deserialize related functions returns pointer into the original byte stream (e.g. this, used here). Personally, I'd expect deserialization to make a copy instead. Thus we'd like to confirm whether the current behavior is intentional?
6. The current library code relies heavily on variable demotion optimization passes to "fix" certain type casts. For instance, the code here wouldn't make sense (and would fail to compile) if we disable the demotion optimizations. While this doesn't lead to immediate issues, if we were to allow disabling optimization passes in the future, it might break the compiler. Thus we'd like to suggest to either explicitly convert between pointer / values within the sway code (e.g. something like this), or move the required demotions into the ir generation phase.
7. Can you point us to the code responsible for encoding witness data for tx? We are unsure whether treating the first 7 bytes of witness data as padding here is appropriate. The reason for our doubt is that the fuel-vm encodes single bytes by adding paddings after the data (not before). While we recognize that these encoding schemes are for different stuff, the usage of different encoding still seems like an easy place for coding mistakes.
8. The existence of both new / old encodings seems pretty confusing, especially for parameter fetching. For instance, if a contract is expecting the old encoding, but callers calls with new encoding, the contract may incorrectly treat the pointer as the function selector. We want to confirm whether the old encoding is required for backward compatibility, and if yes, we'd further recommend adding more documents explaining the difference to prevent misuse.

Findings

Signature verification (fuel-vm)

Input signature verification uses `secp256k1`, thus this should be `eck1()`

Conversion from Bytes64 to PublicKey (fuel-vm)

Conversion from `Bytes64` to `PublicKey` here seems incorrect. `sec1` includes 1 tag byte at the very front of point encoding, so a valid uncompressed `PublicKey` should have 65 bytes.

Tx validity checks (fuel-vm)

A. The document mentions `OutputType.Message` several times (1), (2), (3), but `OutputType.Message` does not exist in the code (should be `Input.MessageOut`)
B. For `TransactionUpgrade`, the document specifies that either `InputType.Coin.owner == PRIVILEGED_ADDRESS` or `InputType.Message.owner == PRIVILEGED_ADDRESS` must be true to validate the tx, but code does not recognize `InputType.Message.owner == PRIVILEGED_ADDRESS`
C. There are some mentions to `scriptLength * 4 != len(script)` and `predicateLength * 4 != len(predicate)` in the document (1), (2), (3), which doesn't seem to match how serialization is done
D. The rule `More than one input of type InputType.Coin for any Coin ID in the input set` doesn't seem to be checked within the code.
E. The document sets `intrinsic_gas_fees` to 0 for `TransactionUpload` and `TransactionUpgrade`, while implementation calculates it in the same way as `TransactionCreate` and `TransactionScript`

Incorrect memory check

We also found an incorrect memory check — I've attached a proof of concept that demonstrates overflowing into a heap region owned by the caller contract.
we'll go through and do variant analysis on this particular class of issue, but we also think this could've been caught with a testcase. would it make sense to have explicit negative tests for memory violations, similar to our proof of concept?

LDC issues

Here are a few issues related to `LDC`.

1. Update size should be done to the code_size field, and not fp which points to `CallFrame.to` here
2. The size of code_size is `WORD_SIZE`, not `CallFrame::code_size_offset().saturating_add(WORD_SIZE);` here.
3. The calculation of `new_code_size` does not consider paddings in the original code, which may lead to incorrect sizes being recorded in `CallFrame`.
4. The spec states that `LDC` could be used to "concatenate" code of multiple contracts. Currently the opcode does `orig_contract.code || orig_contract.padding || new_contract.code || new_contract.padding` instead of `orig_contract.code || new_contract.code || padding`. We think this appears confusing, and want to confirm whether it is intended. Additionally, if it's intended, it might be necessary to highlight this in the documents to prevent misuse.

Other Fuelvm issues

1. When updating `Output`, what is expected to happen if an `Output::Change` does not exist for assets with remaining balance? A related problem is that the team previously mentioned that `fuel-vm panics` are acceptable since it is caught by `fuel-core`, but in case of `panic`, is the gas fee consumed by vm before `panic` still collected? Or is the transaction simply discarded? If it's the latter case, this might expose the node to DoS risks, where malicious actors may repeatedly submit transactions that panic to degrade overall transaction throughput.
2. For `smo` in an external context, is it intended to use `tx_id` as the `sender`?
3. The `flag` register in fuel is a powerful construction that allows users to tune the behavior of arithmetic operations. However, the current implementation makes callee contracts inherit the `flag` set by caller contract. This may become a major footgun for unsuspecting developers. We recommend to either have the compiler generate code to clear `flag` on the contract entry, or have the vm set `flag` to 0 for contract calls.

Optimization passes of sway findings

Issues

1. Loops involving entry block are not properly handled. This may have many kinds of consequences, ranging from compiler panics to incorrect optimization. Notably, we are not yet able to confirm whether or not this scenario is reachable from compiling sway code, given that we haven't been able to have `sway -> ir` conversion produce ir with branches to entry block (but this is merely an observation since we haven't reviewed the compiler frontend, and also are not confident enough to claim optimizations would never produce such code). We are reporting this issue since we believe a proper optimization pass should not modify code execution result as long as there are no undefined behaviors, and branches to entry doesn't really seem like an UB to us.
    a. Dominance frontier analysis may panic if entry block has more than one in edge
    b. Infinite loops may occur in `simplify_cfg` while chaining blocks
    c. `mem2reg` may incorrectly optimize ir and produce code that executes to different results.
2. `inline` pass is unable to handle recursive (self-calling) functions due to it modifying the caller block before fetching instructions to inline within callee block.
3. I believe we should use `checked_mod` for the Mod operations to avoid panics. This also makes sense in the context of fuel since the flag may allow division by 0 to not abort execution.

---

1. We are a bit unsure about how `fndedup` hashes functions. To be clear, we haven't PoCed the concerns listed, but could try to do so if the team thinks it's necessary.
    a. `FxHasher` is currently used for function hashing, which is not a cryptographically secure hash according to the document. Diving deeper into the hasher

implementation reveals that there are no collision / pre-image protections. The potential impacts of this are
    i. If two function hashes collide, it would be catastrophically for the contract being compiled
    ii. A more obscure effect is if malicious actors may intentionally craft contracts that have different functions deduped, which could potentially lead to confusion / losses for contract users
b. Function hashing iterates over fields that are of dynamic length, which means it might be susceptible to malleability issues. To be more precise, the result of `a.hash(hasher); b.hash(hasher)` could be identical to `c.hash(hasher); d.hash(hasher); e.hash(hasher)` if `a || b == c || d || e`. The current usage of discriminator within hash function doesn't really mitigate the problem since misaligned structures would always provide chances for discriminators to match with some different fields / structures. Malleability has been one of the major concerns for hashing in blockchain (e.g. merkle tree leaves / nodes), and we believe it would be better to redesign the hashing scheme to alleviate the risk.

2. Several passes rely on the `escape_analysis` here. Unfortunately, we've noticed flaws within the implementation which could lead to incorrect optimization. Taking `dce` as example, due to imprecise symbol tracking, the attached PoC script would be incorrectly optimized, and the `store` in `main` will be removed. The immediate cause seems to be `load / store / escape` symbol resolve ignoring the `Incomplete` tag of `ReferredSymbols` (1) / (2) / (3), but it is also reasonable to attribute this to incorrect instruction effect modeling. An easy fix would be making all optimizations more conservative (e.g. give up when escaped symbol is `Incomplete`), however, this might limit the effectiveness of optimization passes. Notably, instruction modeling is a complex topic which has been a major source of bugs in compilers (e.g. v8 jit), and even one incorrect effect modeling would lead to failure of the entire algorithm. While the ir instruction set is a lot more limited in `sway`, it would still be really difficult for us to guarantee correctness if algorithm is not sufficiently conservative.
3. The `cei-pattern-analysis` doesn't consider `mint` / `burn` as "effects". This is not the intended behavior, right?
4. Why is the `prepare_sign` skipped for `Create` transactions in `prepare_init_execute` here?
5. We've been looking at gas accounting within the fuel-vm, specifically checking for operations that undercharge / charge incorrectly
    a. `load_contract_code` charges gas after loading the contract, it should first query its length, charge gas and only then load it in memory. Depending on storage implementation, this could be relevant since users may be able to perform costly actions without paying.
    b. Incorrect gas charged for `wdop` / `wqop` operations (1) / (2)

c. `cfei` / `cfe` / `aloc` deal with `Vec` memory allocation and memory zeroing / copying, but only charges a constant gas instead of linear to memory allocation size (1) / (2) / (3)
d. Some opcodes iterate `input` / `output` to find specific entries, but doesn't charge linearly to `input` / `output` count. Given that `input` / `output` count is relatively limited, it is possible that this is absorbed into the constant gas cost. But we still want to confirm whether this is the case or not. (1) / (2) / (3) / (4) / (5) / (6) / (7) / (8)

Sway library findings

1. It seems that the arithmetics operations are intended to be "safe" (i.e. panic on overflow / illegal operations). However, this is not guaranteed for `u64` / `u256` `add` / `mul`, as well as `sub` / `div` / `mod` for all primitive integer types. For instance, the add here only panics when `F_WRAPPING` is not set.
2. `U256` ops sometimes fetches values from incorrect address. We examined the generated ir and it seems correct, thus the bug most likely resides in the compiler backend (`ir -> fuel` conversion). We haven't reviewed that part of the code, but still raising here to let the team know about the bug.
3. `U256.log` implementation may return incorrect values due to rounding errors.
4. We are unsure whether `pow` should be a "safe" operation and panic on overflows. On top of this, the current implementation may return results where `value > type::max`, due to all primitive types of size `< 8 bytes` being treated as `u64`.
5. `Vec` and `Bytes` both performs an out of bound read on `clear`. In the worst case this may lead to unexpected reverts. (1) / (2)
6. `StorageVec.load_vec / store_vec` does not properly handle element types with size `< 8 bytes`. (1) / (2)
7. `StorageMap.remove` should use `sha256(key, self.field_id())` instead of `sha256(key, self.slot())` as key.
8. `b256.try_from(Bytes)` should check against `b.len() != 32` instead of `b.len > 32`
9. `mlog` / `mroo` should be safe operations and panic on illegal inputs. (1) / (2)
10. The uint width for `MAX_INPUTS` / `MAX_WITNESS` / `MAX_MESSAGE_DATA_LENGTH` / `MAX_PREDICATE_LENGTH` / `MAX_PREDICATE_DATA_LENGTH` does not match between fuel-vm / sway / spec. There probably isn't any immediate impact due to the configurations being set within the range of the smallest used type, but we'd recommend unifying those to prevent future issues. (1) / (2) / (3) / (4) / (5). (vm impl)
11. We're wondering whether it makes sense to use `Identity` instead of `Address` for `input_message_sender` and `input_message_recipient`. (1) / (2)
12. Currently, sway `Output` does not recognize the `ContractCreated` variation, is this intentional?
13. We are wondering whether it makes sense to support `Output::Change` (and maybe `Output::Variable` too) for `output_asset_id` / `output_asset_to`. (1) / (2)

14. We are a bit curious about the `caller_address` function. Theoretically it should be possible for a tx to include no `Input::Coins` or `Input::Coins` from multiple senders. In both case, the `caller_address` function would fail to resolve the `caller`. Additionally, the `unwrap` here would revert for `Input::Message`. We want to confirm whether these are intentional.
15. `predicate_address` reverts for `Input::Message`, is this the intended behavior?
16. In `input_message_data`, we should subtract the `offset` from `length` before further usage.
17. I believe we should dereference the pointer in `code_size` here.