PUBLIC

Limited Code Review

of the Vyper Compiler Built-ins and Bytecode Generation

December 13, 2023

Produced for

by

Contents

1 Executive Summary 3
2 Review Overview 5
3 Limitations and use of report 9
4 Terminology 10
5 Findings 11
6 Informational 18

1 Executive Summary

Dear Vyper team,

Thank you for trusting us to help Vyper with this review. Our executive summary provides an overview of subjects covered in our review of the latest version of Vyper Compiler according to Scope to support you in forming an opinion on their security risks.

Limited code reviews are best-effort checks and don't provide assurance comparable to a non-limited code assessment. This review was not conducted as an exhaustive search for bugs, but rather as a best-effort sanity check for the pull requests of interests. The review was executed by one engineer over a period of two weeks. Given the large scope and codebase and the limited time, the findings aren't exhaustive.

The subjects covered by our review are detailed in the Review Overview section.

The large number of issues related to the behavior of the compiler if the builtin functions are folded or not shows that special attention should be given to this part of the compiler. We find that the ongoing effort of merging the general Vyper semantics and folding semantics is the right approach to solve those issues altogether.

The general subjects covered are memory allocation and safety, order of evaluation and semantics of the builtin functions. No major issue was found in the aforementioned subjects.

The following sections will give an overview of the system and the issues uncovered. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1
Low-Severity Findings	17

2 Review Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The review consisted of a non-exhaustive general review of `vyper.builtins` and `vyper.ir.compile_ir` and was conducted within the available time constraints.

This review was not conducted as an exhaustive search for bugs, but rather as a best-effort sanity check. The issues already documented on the `https://github.com/vyperlang/vyper` repository or in older reports at the time of the review were not included in this report.

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	01 December 2023	cbac5aba53f87b388e08f169481d6b5c29002c27	Initial Version

2.1.1 Excluded from scope

All other files and imports that were not mentioned in Scope.

2.2 System Overview

The Vyper language is a pythonic smart-contract oriented language, targeting the Ethereum Virtual Machine (EVM). The Vyper compiler translates the Vyper language into the EVM bytecode. The compilation process is performed in multiple phases:

1. Vyper Abstract Syntax Tree (AST) is generated from Vyper source code.
2. Literal nodes in the AST are validated.
3. Constants are replaced in the AST with their value and constant expressions are folded.
4. Module-level objects are added to the namespace and top-level declarations are validated.
5. Getters for public variables are added, and unused statements are removed from the AST.
6. The semantics of the program are validated. The structure and the types of the program are checked and type annotations are added to the AST.
7. Positions in storage and code are allocated for storage and immutable variables.
8. The Vyper AST is turned into a lower-level intermediate representation language (IR).
9. Various optimizations are applied to the IR.
10. The IR is turned into EVM assembly.
11. Assembly is turned into bytecode, essentially resolving symbolic locations to concrete values.

We now give a brief overview of the two main components we are interested in: the semantic validation and the code generation.

2.2.1 Semantic Validation

Once the Abstract Syntax Tree has been folded, it is analyzed to ensure that it is a valid AST regarding Vyper semantics and annotated with types and various metadata so that the code generation module can properly generate IR nodes from the AST.

The semantics validation starts with the `ModuleAnalyzer` which iterates over the various Module-level statements of the contract. For each statement, after performing various checks, the compiler updates the namespace to add a new entry if needed. For example, for a variable declaration, the namespace will be updated to map the variable's name to some data structure with relevant information such as its type, whether it is public or constant for example.

Once all module-level statements have been properly analyzed, the compiler checks some properties of the module, for example, whether there are no circular function calls or collisions between function selectors.

At this point, getters for public variables are added to the AST and unused statements are removed from it, this includes for example constant variable declaration, which is no longer needed as the constant has been inlined and the declaration has been validated.

The `FunctionNodeVisitor` is then used to validate the content of each function one by one. It iterates over all the statements in the body of the function, and, for each statement, validates that it respects Vyper semantics and, if needed, calls functions like `validate_expected_type` or `get_possible_types_from_node` to analyze the type of the statement's sub-expressions.

The `_ExprAnalyser` is used to infer one or multiple types for a given expression. Such a process can be recursive in the case of complex expressions and mostly acts as a type checker.

The `FunctionNodeVisitor` is also in charge of validating several properties of the functions, for example, that its body respects the function's mutability, or that there is eventually a terminus node at the end of the function if it has a return type.

Once the full Vyper contract has been validated, the `StatementAnnotationVisitor` and the `ExpressionAnnotationVisitor` are called to finally annotate the expression nodes of the AST with their corresponding type for the code generation that will happen later.

2.2.2 Code Generation

After the Abstract Syntax Tree has been type-checked, storage slots have been assigned to storage variables, and data locations have been assigned to immutable variables, the resulting AST is forwarded to the code generation phase to be turned into an intermediate representation of the code (IR). The intermediate representation is a lower-level description of the same program, where the operations performed are more similar to the EVM primitives. As such, it handles pointers directly, explicitly performs memory and storage stores and loads, and translates every high-level Vyper concept into an EVM-compatible equivalent. It differs from the assembly because it has some high-level convenience functionality, such as performing conditional jumps with the `if` operator, looping with the `repeat` operator, defining and caching stack variables with the `with` operator and setting new values for them with the `set` operator, marking jump locations with the `label` operator. Furthermore, it has some convenience functions such as `sha3_32` and `sha3_64`, which use the keccak hash function to compute the hashes of stack variables and the `deploy` function, which copies the runtime bytecode to memory and returns it at the end of the constructor execution.

The code generation is accessed from the function `vyper.codegen.module.generate_ir_for_module()`, which accepts a `GlobalContext` containing the annotated AST as well as some properties such as variables and immutables information. Code is generated for the runtime (code that will be returned by the smart contract constructor and stored in the smart contract) and for the constructor/deployment. Functions are sorted topologically according to the call graph, code is generated first for functions that don't call other functions, then for functions that depend on those, and so on. The memory allocation strategy for a function is to reserve a memory frame

large enough for every callee at the beginning of the function memory frame, and then allocate the variables after the biggest memory offset used by any of the callees.

2.2.2.1 Deployment code and constructor

In the case there is no explicitly defined constructor in the contract, the deployment code is rather straightforward, it simply copies the runtime bytecode from the the deployment code itself to the memory before returning both the offset in memory where the runtime bytecode starts and its length. This operation is abstracted away by the IR using the `deploy` pseudo-opcode.

If there is a constructor defined, the compiler assumes that the compiler arguments, if there are any, will be appended to the deployment bytecode. At the IR level, the `dload` pseudo opcode can be used to read such arguments. The immutables assigned in the constructor, if any, must be returned by the deployment bytecode together with the runtime bytecode to be accessible at runtime. To append them at the end of the runtime bytecode, the IR offers the `istore/iload` abstractions which essentially perform `mstore/mload` at the index corresponding to the end of the buffer for the runtime bytecode in the memory. The runtime bytecode can access them at the IR level with the pseudo-opcode `dload`, which, similarly to its use in the deployment context with constructor arguments, performs `codecopy` from the bytecode's immutable section (at the very end of the bytecode) to the memory. Once the logic of the constructor has been executed, the runtime bytecode concatenated with the immutables is returned using the `deploy` pseudo-opcode.

2.2.2.2 External function and entry points

In the following, an entry point can be seen as an external function if the function has no default arguments. If the function has some, there exists one entry point for each combination of calldata arguments/default overridden argument that is possible. In other words, A function with `D` default argument will have `D+1` entry points. An external function with keyword arguments will generate several entry points, each setting the default values for keyword arguments, and then calling a common function body.

The structure of the runtime bytecode depends on the optimization that has been specified when compiling the contract.

2.2.2.3 Linear selector section

When no optimization is chosen (`optimize=none`), the compiler will generate a linear function selector as it used to do in its previous versions. This selector section acts as follows:

- If the `calldatasize` is smaller than 4 bytes, the execution goes to the fallback.
- For each entry point `F` defined in the contract, the bytecode checks whether the given method id `m` in the calldata matches the method id of `F`.
    - If it does, several checks are performed, such as ensuring `callvalue` is null if the function is non-payable or making sure that `calldatasize` is large enough for `F`. If all the checks pass, the body of the entry-point is executed, otherwise, the execution reverts.
    - If it does not, the iteration goes to the next entry point and, if it was the last one, to the fallback.

2.2.2.4 Sparse selector section

When the gas optimization is set (`optimize=gas`), at compile time, the entry points are sorted by buckets. The amount of bucket is roughly equal to the amount of entry points but can differ a bit as the compiler tries to minimize the maximum bucket size. An entry point with a method id `m` belongs to the bucket `i = m % n` where n is the number of buckets. Once all the buckets are generated, a special data section is appended to the runtime bytecode, it contains as many rows as there are buckets, all rows have the same size and row `k` contains the code location of the handler for the bucket `k`. In the case that the bucket was to be empty, its corresponding location is the fallback function.

When the contract is called, the method id `m` is extracted from the calldata. Given the code location of the data section `d`, the size of its rows (2 bytes) and the bucket to look for (`i = m % n`), the location of the handler for the bucket `i` is stored at location `d + i * 2`. The execution can then jump to the bucket handler location. Very similarly to the non-optimized selector table described above, each bucket handler essentially is an iteration over the entry points it contains, trying to match the method ID. If one of them matches, some `calldatasize` and `callvalue` checks are performed before executing the entry point's body. In the case none of the bucket's entry points match `m`, the execution goes to the `fallback` section.

2.2.2.5 Dense selector section

If the codesize is optimized (`optimize=codesize`), the selector section is organized around a two-step lookup. First, very similarly to the sparse selector section, the method ID can be mapped to some bucket ID `i` which can be used as an index of the `Bucket Headers` data section to read `i`'s' metadata. For a given bucket `b` with id `i` of size `n`, the row `i` of the `Bucket Headers` data section contains `b`'s magic number, `b`'s' data section's location as well as `n`. The bucket magic `bm` is a number that has been computed at compile time such that `(bm * m) >> 24 % n` is different for every method ID `m` of entry-point contained in `b`. Having this unique identifier for methods belonging to `b` means that we can index another data section, specific to `b`. For each entry-point `m` of `b`, this data section contains `m` (its method ID), the location of the entry point's handler, the minimum `calldatasize` it requires accepts and whether or not the entry point is non-payable.

Given all those meta-data, the necessary checks can be performed and the execution can jump to the entry point's body code.

2.2.2.6 Internal and external functions arguments and return values

Function arguments for internal functions are allocated as memory variables at the beginning of the function memory frame. The caller will set their value, accessing the callee memory frame. For external functions, calldata is copied to memory if clamping is needed or if the internal Vyper representation is different than the ABI encoding. Clamping is necessary for types that could exceed their allowed range, such as `uint128`, and ABI encoding and Vyper memory representation differ for dynamic types, for which the ABI encoding includes relative pointers. Return values for internal functions are copied to a buffer allocated in the caller function memory frame. The caller passes on the stack the address of the return buffer to the callee function. The return program counter, for internal functions, is also pushed on the stack by the caller.

2.2.2.7 Function body IR generation

Code for the function body is then generated by calling `vyper.codegen.stmt.parse_body()`. It generates the codes for every statement in a function. Sub-expressions in every statement are recursively parsed. For every type of AST node representing a statement, a function `parse_{NodeType}` is present in `stmt.py`, which generates the IR for a node of type `NodeType` (e.g. `parse_Return`, `parse_Assign`). Expressions contained in statements are recursively parsed in the `vyper.codegen.expr` module. Code is generated for the innermost expressions, and the output of the generated code is used to evaluate the containing expressions.

2.2.3 Built-in functions

There are various built-in functions in Vyper, they can either be statements (do not return a value and cannot be assigned) or expressions. In general, built-ins offers feature that couldn't be implemented in plain Vyper (for example access to some opcodes like `CREATE` or `CREATE2`). During semantic validator, several properties are checked (the arguments given to the calls have types that are valid for the given builtin, if it is required, that certain arguments are constant...). Later, during the code generation phase, the call is converted to some intermediate presentation as implemented in the `build_IR` function of the given built-in.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe our findings. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- extract32's Input Is Not Cached

Low-Severity Findings	17

- Assertion Error _abi_encode With Invalid Method ID
- Buffer Too Large in create_minimal_proxy_to
- Builtins Fail to Compile With Empty ByteStringT
- Condition Always True in raw_call
- Delegate or Static raw_call With Value
- Expression Builtins Are Treated as Statements
- Imprecise Out-Of-Bounds Check for slice
- Inconsistent Folding of keccak256
- Inconsistent Folding of len
- Inconsistent Folding of shift
- Incorrect Error Message
- Incorrect Type Checking for slice
- Redundant Validation _abi_decode
- Unused Allocated Memory in raw_call
- extract32 and slice With Transient Storage
- shift's Incorrect Return Type
- sqrt Colliding Fresh Variables

5.1 `extract32`'s Input Is Not Cached

Correctness Medium Version 1

CS-VYPER_DECEMBER_2023-001

The input `b` of the built-in `extract32` is not cached but instead, only its location and length are. This means that if the evaluation of the argument `start`'s side effect leads to the modification of `b`, the result of `extract32` could be inconsistent and access memory out of bounds.

For example, calling the function `foo` of the following contract returns `b'uuuuuuuuuuuuuuuuuuuuuuuuuuuu\x00\x00789'`

```
var:Bytes[96]

@internal
def bar() -> uint256:
    self.var = b'uuuuuuuuuuuuuuuuuuuuuuuuuuuuuu'
    self.var = b''
    return 3

@external
def foo() -> bytes32:
    self.var = b'abcdefghijklmnopqrstuvwxyz123456789'
    return extract32(self.var, self.bar(), output_type=bytes32)
    # returns b'uuuuuuuuuuuuuuuuuuuuuuuuuuuu\x00\x00789'
```

5.2 Assertion Error `_abi_encode` With Invalid Method ID

Design | Low | Version 1

CS-VYPER_DECEMBER_2023-002

If `_abi_encode` is given a literal method ID that is not 4 bytes long, the compiler will fail on an assertion in `_parse_method_id` during code generation instead of failing at type-checking time with some meaningful error message.

The following contract would fail to compile with an `AssertionError`:

```
@external
def foo():
    a:Bytes[68] = _abi_encode(b'', method_id=b'')
```

5.3 Buffer Too Large in `create_minimal_proxy_to`

Design | Low | Version 1

CS-VYPER_DECEMBER_2023-003

In `CreateMinimalProxyTo._build_create_IR()`, the buffer `buf` uses 128 bytes of memory, however, `MSTORE` is performed at `buf`, `buf + 19` and `buf + 39`. This means that a buffer of length 96 would be sufficient to store the whole creation bytecode.

Note that although the bytecode stored in the buffer is only 54 bytes long, a buffer of 64 bytes is not sufficient because the last `MSTORE` is unaligned and would write to the next 32-byte word.

5.4 Builtins Fail to Compile With Empty `ByteStringT`

Design | Low | Version 1

CS-VYPER_DECEMBER_2023-004

In general, `BytesT` and `StringT` IR nodes are assumed to be locations to values of the mentioned types. However, the special value `empty(...)` is not and is treated as a special case by the compiler.

The compiler fails compilation when the given builtins are given empty `ByteString`: `concat`, `extract32` and `slice`.

The compilation of the following three contracts respectively fails with the following errors:

- `CompilerPanic: tried to modify non-pointer type`
- `AssertionError: ~empty <empty(Bytes[32])>`
- `AttributeError: 'NoneType' object has no attribute 'load_op'`

```
@external
def baz() :
    a:String[32] = concat(empty(String[31]), "a")
```

```
@external
def foo() :
    a:Bytes[32] = slice(empty(Bytes[32]), 0, 1)
```

```
@external
def bar() :
    a:uint256 = extract32(empty(Bytes[32]), 0, output_type = uint256)
```

Note that for both `extract32` and `slice`, if it wasn't for this issue, either the compilation would fail, or the execution would revert as slicing/extracting 32 bytes from an empty byte array is not possible.

5.5 Condition Always True in `raw_call`

Design | Low | Version 1

CS-VYPER_DECEMBER_2023-005

In `RawCall.fetch_call_return()`, the following `if` block is always evaluated since if `outsize` is `None` or `0`, the function is returning earlier.

```
if outsize.value:
    return_type = BytesT()
    return_type.set_min_length(outsize.value)

    if revert_on_failure:
        return return_type
    return TupleT([BoolT(), return_type])
```

5.6 Delegate or Static `raw_call` With Value

Design | Low | Version 1

CS-VYPER_DECEMBER_2023-006

Although both EVM's `delegatecall` and `staticcall` opcode cannot be given some value to be sent with the call, `raw_call` does not prevent having a non-null value when `is_delegate_call` or `is_static_call` is set to true. Note that the value is then ignored when generating IR/bytecode for the call.

5.7 Expression Builtins Are Treated as Statements

Design | Low | Version 1

CS-VYPER_DECEMBER_2023-007

The following builtins are in `STMT_DISPATCH_TABLE` although they cannot be used as statements since they return a value:

- `create_minimal_proxy_to`
- `create_forwarder_to`
- `create_copy_of`
- `create_from_blueprint`

Although using them as statements will currently raise a `StructureException`, it would be better to remove them from `STMT_DISPATCH_TABLE` altogether.

5.8 Imprecise Out-Of-Bounds Check for `slice`

Design | Low | Version 1

CS-VYPER_DECEMBER_2023-008

In the function `Slice.fetch_call_return`, when `is_adhoc_slice` is false and `start_literal` is not `None`, the following check is performed:

```
if start_literal > arg_type.length:
    raise ArgumentException(f"slice out of bounds for {arg_type}", start_expr)
```

To be more precise, one could also raise if `start_literal == arg_type.length`.

The following contract currently compiles (but calling `foo()` reverts) although the slice can be inferred to be out of bounds at compile time:

```
@external
def foo():
    a:String[3] = "foo"
    l: uint256 = 1
    b:String[3] = slice(a,3,l)
```

5.9 Inconsistent Folding of keccak256

Design Low Version 1

CS-VYPER_DECEMBER_2023-009

Although the built-in `keccak256` function should only accept `StringT`, `bytes32` and `BytesT` types, it also accepts `bytesN` where `N!=32` provided that the node is folded. In other words, the semantics of the built-in differ when it is folded or not.

```
@external
def foo():
    v:bytes1 = 0x12

    a:bytes32 = keccak256(0x12) # compiles
    b:bytes32 = keccak256(v) # does not compiles
    c:bytes32 = keccak256(0x1111111111111111111111111111111111111111111111111111111111111111) # compiles
```

5.10 Inconsistent Folding of len

Design Low Version 1

CS-VYPER_DECEMBER_2023-010

The built-in `len` can be folded when given some literal `Hex` node but the compilation fails if given some non-literal `bytesN` as the type is not accepted.

```
@external
def foo():
    a:uint256 = len(0x12) # compiles
    x:bytes1 = 0x12
    b:uint256 = len(x) # does not compile
```

Moreover, no restriction is enforced on the length of such literal `Hex` nodes when used in `len`, which could mean that there is a `Hex` value for which no Vyper type can be inferred (The program would not compile if it wasn't for folding).

```
@external
def foo():
    # length of 35 is not allowed for bytesN
    a:uint256 = len(0x11111111111111111111111111111111111111111111111111111111111111111111)
```

5.11 Inconsistent Folding of shift

Design Low Version 1

CS-VYPER_DECEMBER_2023-011

The built-in `shift` only accepts non-negative values `x` when folding is performed but in the case the operation is done at runtime, any (negative) `int256` value is accepted.

5.12 Incorrect Error Message

Design Low Version 1

CS-VYPER_DECEMBER_2023-012

`Extract32.infer_kwarg_types()` may raise an `InvalidType` exception if `output_type` is not valid, however, the message says `"Output type must be one of integer, bytes32 or address"` although any `bytesX` is valid.

5.13 Incorrect Type Checking for `slice`

Correctness **Low** **Version 1**

CS-VYPER_DECEMBER_2023-013

When providing to the builtin `slice` a byte array which is an `Attribute` AST node whose attribute is `code` (for example a struct field access), the function `Slice.fetch_call_return` will misunderstand the byte array as `msg.code` (`is_adhoc_slice` evaluates to true), leading to several checks being missed.

In the current state of the compiler, this is not an issue because of #3521_ as the compilation would revert with a `TypeCheckFailure`. However, if the issue is fixed with PR3527, the issue described above would arise.

For example in the following contract, assuming that PR3527 is merged, the declaration of b will compile, but the declaration of c will not.

```
struct A:
code: String[4]
not_code: String[4]

@external
def foo() -> uint256:
    a:A = A({code: "abc",not_code: "abc"})
    b:String[4] = slice(a.code, 1,0) # compiles
    c:String[4] = slice(a.not_code, 1,0) # does not compile
    return b
```

5.14 Redundant Validation `_abi_decode`

Design **Low** **Version 1**

CS-VYPER_DECEMBER_2023-014

In `ABIDecode.infer_arg_types()`, the call to `validate_call_args()` is redundant as `self._validate_arg_types())` is called just before and performs the same validation

```
validate_call_args(node, expect_num_args, list(self._kwargs.keys()))
```

5.15 Unused Allocated Memory in `raw_call`

Design **Low** **Version 1**

CS-VYPER_DECEMBER_2023-015

In the case `raw_call` is given `max_outsize=0`, the function `RawCall.build_IR` still allocates 32 bytes for `output_node`. This is not necessary and could be avoided.

5.16 `extract32` and `slice` With Transient Storage

Design | Low | Version 1

CS-VYPER_DECEMBER_2023-016

In the builtins `extract32` and `slice`, given that storage is addressed in words, some custom logic is implemented when the given buffer lives in the storage as opposed to other locations that are byte addressable. Since transient storage will also be addressed in words, a similar logic should be implemented for buffer in transient storage.

5.17 `shift`'s Incorrect Return Type

Design | Low | Version 1

CS-VYPER_DECEMBER_2023-017

The class `Shift` defines `_return_type` as `UINT256_T` but it is never used since `BuiltinFunctionT.fetch_call_return()` is overridden by `Shift` to return the type of the argument x which can differ from `UINT256_T`.

5.18 `sqrt` Colliding Fresh Variables

Design | Low | Version 1

CS-VYPER_DECEMBER_2023-018

The built-in `sqrt` calls `generate_inline_function` to compile some piece of Vyper code to IR using a new context. As the context is new, the `fresh_varname` counter is reset and does not follow the counter of the initial context. This can lead to the collision of variables, as shown in the following examples where the loop iterators are colliding and leading to the compiler panicking:

```
@external
def foo() :
# the loop iterator variable of each sqrt's loop collide
a:decimal = sqrt(sqrt(100.0))
```

```
@external
def foo() :
# The loop iterator of sqrt collides with the range iterator
for i in range(10):
    a:decimal = sqrt(100.0)
```

6 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

6.1 EIP1167 Bytecode Sanity Checks

Informational **Version 1**

CS-VYPER_DECEMBER_2023-019

The function `eip1167_bytecode` returns the EIP1167 bytecode as well as a "loader" used to deploy the bytecode. As the resulting bytecode is known and should be constant across Vyper versions, having sanity checks comparing the different outputs of `assembly_to_evm` with hardcoded bytecode could be a good failsafe.

Moreover, the length of different bytecode sections is used (e.g `0x2D`, the sum of the lengths of the `forwarder_pre_asm`, the address and the `forwarder_post_asm`), having sanity check that makes sure that such literal integer is indeed equal to the length of the given bytecode section would increase both readability and maintainability of the code.

6.2 `extract32` Unnecessarily Complex

Informational **Version 1**

CS-VYPER_DECEMBER_2023-020

the built-in `Extract32.buildIR` seems unnecessarily complex and could probably be simplified similarly to the slice `slice`. In particular, for locations that are byte-addressable when `start` is not word aligned, the use of a helper like `copy_bytes` could avoid the need for loading two words and then masking and shifting.