PUBLIC

Limited Code Review

of the Vyper Compiler

Semantic analysis and Code generation

June 13, 2023

Produced for

by

Contents

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.

We did not find any issues in the fixes of the security advisories that were in the scope of this review and can confidently assert that the security advisories have been resolved.

The elimination of the Function Signature class enhances the code's readability and consistency, according to our findings. This removal, enabled by the previous pull request that refactored the type system and the code generation, is one of the last steps in merging the type systems of the semantic analysis and the code generation.

The Journal and its commit/rollback scheme fix the issue with incorrect type checking of loop variables but also allows for future new metadata to be added to the compiler easily. Although one issue was found in its implementation as highlighted by Metadata Journal can rollback incorrectly, this new primitive is a great addition to the compiler as it also fixes a performance issue by caching the list of potential types for nodes.

Special attention should be applied to testing complex expressions with functions calls as sub-expression. As highlighted in various issues such as Multiple evaluations of DST lead to non-unique symbol errors when copying Bytes arrays or DynArrays or Make_setter is incorrect for complex types when the RHS references the LHS with a function call, such complex expressions might be edge cases in the compiler logic and should be part of the testing suite.

Additionally, the large amount of issues related to the new IfExp AST node depicts the importance for the compiler to be more generic in its way to validate the semantics of expressions as currently, some functions must handle the case of several AST nodes in distinct ways as they cannot be handled by the general logic.

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	3
Low-Severity Findings	24

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 was performed on the fixes of the recently published security advisories of the `https://github.com/vyperlang/vyper` repository as well as on several important pull requests:

1. https://github.com/vyperlang/vyper/commit/02339dfda0f3caabad142060d511d10bfe93c520
2. https://github.com/vyperlang/vyper/commit/c3e68c302aa6e1429946473769dd1232145822ac
3. https://github.com/vyperlang/vyper/commit/851f7a1b3aa2a36fd041e3d0ed38f9355a58c8ae
4. https://github.com/vyperlang/vyper/commit/0bb7203b584e771b23536ba065a6efda457161bb
5. https://github.com/vyperlang/vyper/commit/4f8289a81206f767df1900ac48f485d90fc87edb
6. https://github.com/vyperlang/vyper/commit/3de1415ee77a9244eb04bdb695e249d3ec9ed868
7. https://github.com/vyperlang/vyper/pull/3388
8. https://github.com/vyperlang/vyper/pull/3390
9. https://github.com/vyperlang/vyper/pull/3410
10. https://github.com/vyperlang/vyper/pull/3375

In addition to the pull requests mentioned, a non-exhaustive general review of `vyper.semantics` and `vyper.codegen` 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 at the time of the review were not included in this report.

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. 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.
5. Getters for public variables are added, and unused statements are removed from the AST.
6. Positions in storage and code are allocated for storage and immutable variables.
7. The Vyper AST is turned into a lower-level intermediate representation language (IR).
8. Various optimizations are applied to the IR.
9. The IR is turned into EVM assembly.
10. Assembly is turned into bytecode.

The review we conducted revolved around phases 4 to 7 of the above list.

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.

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.

Those functions are using the `_ExprAnalyser` to infer one or multiple types for a given expression. Such a process can be recursive in the case of complex expressions and mostly act 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 typechecked, 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 assembly because it has some high level convenience functionality, such as performing conditional jumps with the `if` operator, looping with the `repeat` operator, defining 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 variable containing the annotated AST, and storage and data indexes. 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. 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. For every function, code is generated and concatenated. For external functions, code for selector matching is prepended. An external function with keyword arguments will generate several selectors, setting the default values for keywords arguments, then calling a common function body. 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.

Code for the function body is then generated by calling `vyper.codegen.stmt.parse_body()`. `parse_body()` generates the codes for every statement in a function. Subexpressions 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.

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	3

- Incorrect Order of Evaluation of Arguments of Builtin Function
- Make_setter Is Incorrect for Complex Types When the RHS References the LHS With a Function Call
- Metadata Journal Can Rollback Incorrectly

Low-Severity Findings	24

- Assertion Could Be More Precise in parse_Binop
- Assertions Are Not Constant
- Calls to State-Modifying Functions in Range Expressions Are Not Caught by the Type-Checker
- Default Arguments Are Treated as Keyword Arguments
- Epsilon Is Not Documented
- IfExp Cannot Be Used in a Subscript
- IfExp Fails at Codegen When Used With Self or Environment Variables
- IfExp Not Annotated When Used as Iterable for a Loop
- Implements Statement Does Not Enforce the Same Indexation of Events
- Imported Contracts Are Not Fully Semantically Validated
- Incorrect Typing of Builtins Whose Return Type Depends on Some of Its Argument's Types
- Incorrect Typing of Loop Iterable When It Is a List Literal
- Incorrect Typing of Raw_Call When Max_Outsize=0 in Kwargs
- Multiple Evaluations of DST Lead to Non-Unique Symbol Errors When Copying Bytes Arrays or DynArrays
- Nesting a Pop in Append Results in Incorrect Behavior
- No Sanity Check on Storage Layout Files
- Overriding Storage Layout Fails With Immutables
- PR 3134 Has Been Reverted
- Redundant and Incomplete Function Selector Collision Checks
- References to Public Constant and Immutables With Self Missed by the Typechecker
- Semantic Analysis of the Imported Contract Is Done With the Current Contract's Namespace

- StateAccessViolation When "Self" Is Used as a Struct Field Name
- TypecheckFailure When Using Address and Self Members as Struct Field Name
- in and Not in Cannot Be Used With DynArray of Enums

5.1 Incorrect Order of Evaluation of Arguments of Builtin Function

Correctness Medium Version 1

CS-VYPER_MAY_2023-001

The order of evaluation of the arguments of the builtin functions `uint256_addmod`, `uint256_mulmod`, `ecadd` and `ecmul` is incorrect.

- For `uint256_addmod(a,b,c)` and `uint256_mulmod(a,b,c)`, the order is `c,a,b`.
- For `ecadd(a,b)` and `ecmul(a,b)`, the order is `b,a`.

In the following contract, a call to `foo()` returns `1` while we would expect it to return `0`.

```
a:uint256

@internal
def bar() -> uint256:
    self.a = 1
    return 8

@external
def foo()->uint256:
    return uint256_addmod(self.a, 0, self.bar()) # returns 1
```

In the following contract, a call to `loo()` returns `False` while we would expect it to return `True`.

```
x: uint256[2]

@internal
def bar() -> uint256[2]:
    self.x = ecadd([1, 2], [1, 2])
    return [1,2]

@external
def loo() -> bool:
    self.x = [1, 2]

    a:uint256[2] = ecadd([1, 2], [1, 2])
    b:uint256[2] = ecadd(self.x, self.bar())

    return a[0] == b[0] and a[1] == b[1] # returns false
```

5.2 Make_setter Is Incorrect for Complex Types When the RHS References the LHS With a Function Call

Correctness Medium Version 1

CS-VYPER_MAY_2023-002

Issue 2418 described a bug where, during an assignment, if the right-hand side refers to the left-hand side, part of the data to be copied may get overwritten before being copied.

Although PR 3410 fixed the issue in most of the cases, it can still happen with function calls as shown in the example below. A call to `foo` returns `[2,2]` where if the function `bar` would be inlined, it would return `[2,1]`

```
a:DynArray[uint256,2]

@external
def foo() -> DynArray[uint256,2]:
    # Initial value
    self.a = [1,2]
    self.a = [self.bar(1), self.bar(0)]
    return self.a #returns [2,2]

@internal
def bar(i:uint256)->uint256:
    return self.a[i]
```

In this second example, `boo` temporarily assigns values to a before emptying it. the values stored in a are however still readable from `foo` as a call to `foo` here returns `[11,12,3,4]`.

```
a:DynArray[uint256, 10]

@external
def foo()->DynArray[uint256,10]:
    self.a = [1,2,self.boo(),4]
    return self.a # returns [11,12,3,4]

@internal
def boo() -> uint256:
    self.a = [11,12,13,14,15,16]
    self.a = []
    # it should now be impossible to read any of [11,12,13,14,15,16]
    return 3
```

5.3 Metadata Journal Can Rollback Incorrectly

Correctness Medium Version 1

CS-VYPER_MAY_2023-003

To fix the issue of incorrect type checking of loop variables, a commit/rollback scheme for metadata caching has been implemented to handle speculation when trying to type a loop.

When registering two consecutive updates for a given node, the journal can have an incorrect behavior.

Assuming that the compiler has entered the speculation mode (while typing a loop for example), and considering an AST node `A` which, at the time of entering the speculation had `M0` as metadata, if the following events happen, the cached metadata for `A` would become incorrect (considering `M0!=M1`):

1. The metadata of `A` is updated a first time (using `register_update`) resulting in `M1`.
2. The metadata of `A` is updated a second time resulting in `M2` (which might or might not be equal to `M1`).
3. `_rollback_inner` is called to roll back `A`'s metadata to its state pre-speculation.

While the correct state of `A`'s metadata should be `M0`, the resulting metadata will currently be `M1` as the second call to `register_update` has "overwritten" the first one.

5.4 Assertion Could Be More Precise in parse_Binop

Design Low Version 1

CS-VYPER_MAY_2023-004

In the function `Expr.parse_BinOp` in the code generation, the assertion `is_numeric_type(left.typ)` could be performed before the `LShift` and `RShift` cases are those operators are only defined for numeric types.

5.5 Assertions Are Not Constant

Correctness Low Version 1

CS-VYPER_MAY_2023-005

The definition of the class `Context` introduce the flag `in_assertion` which, when set, indicates that the context should be constant according to `is_constant()` definition. This flag is never set during the code generation, specifically, it is possible to have a non-constant expression in an `assert` statement. For example, the following contract compiles.

```
x: uint256

@internal
def bar() -> uint256:
    self.x = 1
    return self.x

@external
def foo():
    assert self.bar() == 1
```

5.6 Calls to State-Modifying Functions in Range Expressions Are Not Caught by the Type-Checker

Correctness Low Version 1

CS-VYPER_MAY_2023-006

The type checker does not catch the use of a state-modifying function call in a range expression, this leads the code generator to fail due to an assertion: `assert use_staticcall, "typechecker missed this"`

The compiler fails to compile the following with the assertion mentioned above.

```
interface A:
    def foo()-> uint256:nonpayable

@external
def bar(x:address):
    a:A = A(x)
    for i in range(a.foo(),a.foo()+1):
        pass
```

5.7 Default Arguments Are Treated as Keyword Arguments

Design Low Version 1

CS-VYPER_MAY_2023-007

`validate_call_args` takes `kwargs`, the list of valid keywords as an argument and makes sure that when a call is made, the given keywords are valid according to `kwargs`.

When being called from `ContractFunctionT.fetch_call_return`, the defaults arguments of the function are given to `validate_call_args` in `kwargs` although it is not allowed to give keywords arguments in a function call except for `gas`, `value`, `skip_contract_check` and `default_return_value`.

For example, when trying to compile the following contract, the call to `validate_call_args` made by `fetch_call_return` will succeed although an invalid keyword argument is passed. The compilation will later fail (as it should) as `fetch_call_return` enforce that the kwargs should belong to the call site kwargs' whitelist.

```
@external
def foo():
    self.boo(a=12)

@internal
def boo(a:uint256=12):
    pass
```

5.8 Epsilon Is Not Documented

Design Low Version 1

CS-VYPER_MAY_2023-008

The builtin function `epsilon` is not documented in https://docs.vyperlang.org/.

5.9 IfExp Cannot Be Used in a Subscript

Design Low Version 1

CS-VYPER_MAY_2023-009

The `IfExp` AST node's case is missing in `util.py:types_from_Subscript` and `annotation.py:visit_subscript`. The following example does not compile, and the compiler returns: `vyper.exceptions.StructureException: Ambiguous type`

```
@external
def boo() :
    a:uint256 = ([1] if True else [2])[0]
```

5.10 IfExp Fails at Codegen When Used With Self or Environment Variables

Design Low Version 1

CS-VYPER_MAY_2023-010

Some complex expressions including the new `IfExp` node might typecheck, however, no corresponding case is implemented in the codegen leading the compiler to fail.

The following example fails to compile with an assertion error (`isinstance(contract_address.typ, InterfaceT)`) in `ir_for_external_call`.

```
@external
def foo():
    (self if True else self).bar()

@internal
def bar():
    pass
```

The following example fails to compile with `vyper.exceptions.TypeCheckFailure: Name node did not produce IR`.

```
@external
def foo():
    a:Bytes[10] = (msg if True else msg).data
```

Note: In case the first example was to be allowed by Vyper, one would need to be careful as several analysis and sanity checks (e.g circularity checks) rely on the fact that function calls are always on the form `self.FUNC_NAME`.

5.11 IfExp Not Annotated When Used as Iterable for a Loop

Design Low Version 1

CS-VYPER_MAY_2023-011

The `IfExp` AST node's case is missing in `annotation.py:visit_For` leading the `StatementAnnotationVisitor` to omit the annotation of a `IfExp` node when used as iterable in a loop. The following example does not compile, and the compiler returns: `KeyError: 'type'`.

```
@external
def foo():
    for x in [1,2] if True else [0,12]:
        pass
```

Note that if a new case for `IfExp` is created in `annotation.py:visit_For` to fix this issue, the function `local.py:visit_For` should be updated carefully as the check that ensures that for loops must have at least 1 iteration would not be performed on `IfExp` nodes.

5.12 Implements Statement Does Not Enforce the Same Indexation of Events

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

CS-VYPER_MAY_2023-012

When using the `implements` statement, the contract's events fields are not enforced to match the interface's events fields on their indexation.

For example, the following code compiles although the `spender` field of `Approval` is not indexed.

```
from vyper.interfaces import ERC20

implements: ERC20

event Transfer:
    sender: indexed(address)
    receiver: indexed(address)
    value: uint256

event Approval:
    owner: indexed(address)
    spender: address
    value: uint256

name: public(String[32])
symbol: public(String[32])
decimals: public(uint8)

balanceOf: public(HashMap[address, uint256])
allowance: public(HashMap[address, HashMap[address, uint256]])
totalSupply: public(uint256)


@external
def __init__(_name: String[32], _symbol: String[32], _decimals: uint8, _supply: uint256): pass

@external
def transfer(_to : address, _value : uint256) -> bool: return True

@external
def transferFrom(_from : address, _to : address, _value : uint256) -> bool: return True

@external
def approve(_spender : address, _value : uint256) -> bool: return True
```

5.13 Imported Contracts Are Not Fully Semantically Validated

Correctness Low Version 1

CS-VYPER_MAY_2023-027

When importing a contract, only the function signatures are semantically checked (to produce the `InterfaceT`), A contract that does not compile could be imported in another one which would then compile as long as the signatures of the imported functions are semantically correct.

For example, `a.vy` compiles although it imports `b.vy` which does not compile.

```
#a.vy
import b as B

@external
def foo(addr:address):
    x:B = B(addr)
    x.foo()
```

```
#b.by
@external
def foo():
    x:uint256 = "foo"
```

5.14 Incorrect Typing of Builtins Whose Return Type Depends on Some of Its Argument's Types

Correctness Low Version 1

CS-VYPER_MAY_2023-013

Builtin functions whose return type depends on some of its argument's type can be incorrectly typed resulting in the compiler exiting with a `TypeMismatch`.

To achieve this behavior, the builtin function should be called with arguments such that:

- At least one argument is not constant as the call would be folded otherwise.
- `get_possible_types_from_node` should return multiple potential types for the arguments on which the return type of the builtin depends.

Below is a list of the builtins affected together with examples failing to compile although they should:

- `min` and `max`:
    - `a:uint256 = min(1 if True else 2, 1)`
- all unsafe builtins:
    - `a:uint256 = unsafe_add(1 if True else 2, 1)`
- `shift` (deprecated as of v0.3.8):
    - `a:uint256 = shift(-1, 1 if True else 2)`
- `uint2str`:

- `f:String[12] = uint2str(1 if True else 2)`

5.15 Incorrect Typing of Loop Iterable When It Is a List Literal

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

CS-VYPER_MAY_2023-014

when a loop iterates over a literal list, the function `visit_For` of the `StatementAnnotationVisitor` annotates it with a Static Array type whose value type is the last element of the list of common types of shared by the elements. To be consistent with the previously performed analysis, the list should be typed using the type of the loop iterator as it is done with `range` expressions.

In this code, although it compiles, `i` is typed as a `uint8` while `[1,2,3]` is annotated with `int8[3]`.

```
@external
def foo():
    for i in [1,2,3]:
        a:uint8 = i
```

When doing the code generation of a `for` loop iterating over a literal list, `_parse_For_list` is overwriting the value type of the list with the type of the loop iterator inferred at type checking. This behavior is commented with: `TODO investigate why stmt.target.type != stmt.iter.type.value_type`. By solving the issue above, `stmt.target.type` would be equal to `stmt.iter.type.value_type` and no overwriting would be needed.

5.16 Incorrect Typing of Raw_Call When Max_Outsize=0 in Kwargs

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

CS-VYPER_MAY_2023-015

When called with `max_outsize` explicitly set to `0` (`max_outsize=0`) the compiler wrongly infers that `raw_call` has no return type.

```
@external
@payable
def foo(_target: address):

    # compiles
    a:bool = raw_call(_target, method_id("someMethodName()"), revert_on_failure=False)

    # does not compile but should compile
    b:bool = raw_call(_target, method_id("someMethodName()"), max_outsize=0, revert_on_failure=False)

    # compiles but should not compile
    raw_call(_target, method_id("someMethodName()"), max_outsize=0, revert_on_failure=False)
```

5.17 Multiple Evaluations of DST Lead to Non-Unique Symbol Errors When Copying Bytes Arrays or DynArrays

Correctness Low Version 1

CS-VYPER_MAY_2023-016

The destination of byte arrays and DynArray copying is evaluated multiple times as `cache_when_complex` is not used. This includes the following functions:

- `make_byte_array_copier`.
- `_dynarray_make_setter` (both cases: `src.value == "multi"` and `src.value != "multi"`).

For example, compiling the following Vyper code will output `AssertionError: non-unique symbols {'self.bar()2'}`.

```
a:DynArray[DynArray[uint256, 2],2]

@external
def foo():
    self.a[self.bar()] = [1,2]
@internal
def bar()->uint256:
    return 0
```

5.18 Nesting a Pop in Append Results in Incorrect Behavior

Correctness Low Version 1

CS-VYPER_MAY_2023-017

When modifying the size of a DynArray during a call to `append`, the initial length will be the one used to compute the new length and the compiler won't consider any change of length done by the sub-expression. In the example below, the value returned by `a.pop()` is used but its side effect of decreasing a's length is omitted.

This behavior was introduced by the fix to the security advisory OOB DynArray access when array is on both LHS and RHS of an assignment. As the length of the `append` is cached before the evaluation of the `pop` and stored in memory after, the new length produced by the `pop` which is stored in the memory is not taken into account as it is overwritten by the cached length.

```
@external
def foo() -> DynArray[uint256,3]:
    a:DynArray[uint256,2] = [12]
    a.append(a.pop())
    return a # outputs [12,12] while the same in python outputs [12]
```

5.19 No Sanity Check on Storage Layout Files

Design Low Version 1

CS-VYPER_MAY_2023-018

When compiling a contract with the flag `storage_layout_file`, some basic sanity checks could be performed on the given JSON file as currently:

- The JSON can have duplicated entries. In this case, the last one will be the one used by the compiler.
- The JSON can have entries not matching any storage slot of the contract
- The entries of the JSON do not necessarily have to match with the type of the corresponding variables in the contract.

For example, a contract only defining the storage variable `a:uint256` can be compiled given the following storage layout:

```
{
    "a": {"type": "uint16", "slot": 10},
    "a": {"type": "uint8", "slot": 1},
    "b": {"type": "uint256", "slot": 1}
}
```

5.20 Overriding Storage Layout Fails With Immutables

Correctness Low Version 1

CS-VYPER_MAY_2023-019

The check used for ignoring immutable in `set_storage_slots_with_overrides` is ill-defined.

When compiling a contract with a custom storage layout file, if an immutable is defined in the contract (and is not present in the json), the compilation will fail with a `StorageLayoutException`.

For example, the following contract fails to compile if given an empty storage layout.

```
a:immutable(uint256)

@external
def __init__():
    a = 1
```

5.21 PR 3134 Has Been Reverted

Design Low Version 1

CS-VYPER_MAY_2023-020

The PR 3134 has been reverted by the PR 2974 in the sense that neither the conflicting signatures nor the method ID are displayed when the multiple functions share the same selector.

Note however that PR 2974 fixed an issue where collision between functions having `0x00000000` has method ID were not detected since `collision == 0` would be treated as `collision == Null`.

5.22 Redundant and Incomplete Function Selector Collision Checks

Design Low Version 1

CS-VYPER_MAY_2023-021

To ensure the method IDs are unique, the constructor of the `ModuleAnalyzer` performs two checks that are both incomplete but together cover every case:

- The call to `validate_unique_method_ids` by the constructor of the `ModuleAnalyzer` does not handle the public variable getters as they haven't been added to the AST yet.
- The generation of an `InterfaceT` from the top-level node has as a side effect to ensure the uniqueness of method IDs of public variable getters and external functions but does not handle internal variables (not really required at the moment but in Vyper semantics to prevent breaking changes in case of a future change to their calling convention).

It would probably be better to have one check that covers everything for clarity purposes.

5.23 References to Public Constant and Immutables With Self Missed by the Typechecker

Design Low Version 1

CS-VYPER_MAY_2023-022

As `visit_VariableDecl` adds public constant and immutables variables to `self`'s namespace, `types_from_Attribute` successfully typecheck references to constant and immutables using `self`. The compiler later fails during the codegen.

Compiling the following contract will fail with `KeyError: 'a'`.

```
a:public(constant(uint256)) = 1

@external
def foo():
    b:uint256 = self.a
```

5.24 Semantic Analysis of the Imported Contract Is Done With the Current Contract's Namespace

Correctness Low Version 1

CS-VYPER_MAY_2023-023

When an imported interface is typed, the namespace of the current contract is used to generate the interface type from the AST of the imported contract. This means that the imported contract's function definitions may use the types and constants defined in the current contract.

For example `a.vy` would compile successfully although `b.vy`, which is imported by `a.vy` makes use of `S` and `a`, both defined in `a.vy`.

```
#a.vy
import b as B

struct S:
    x:uint256
a:constant(uint256) = 12

@external
def bar(addr:address):
    x:B = B(addr)
    y:S = x.foo()
```

```
#b.vy
@external
def foo(a:uint256=a) -> S:
    return S({x:12})
```

5.25 StateAccessViolation When "Self" Is Used as a Struct Field Name

Correctness Low Version 1

CS-VYPER_MAY_2023-024

While it is allowed to use `self` as a field name for a struct, constructing such struct in a `pure` function will result in a `StateAccessViolation` as the compiler will consider that this is a reference to `self`, the address of the contract.

For example, the following contract fails to compile due to `StateAccessViolation: not allowed to query contract or environment variables in pure functions`.

```
struct A:
    self:uint256

@external
@pure
def foo():
    a:A = A({self:1})
```

5.26 TypecheckFailure When Using Address and Self Members as Struct Field Name

Correctness Low Version 1

CS-VYPER_MAY_2023-025

Accessing the field of an enum named after an address or self member (`balance`, `codesize`, `is_contract`, `codehash` or `code`) results in a `TypeCheckFailure`.

For example, the following contract fails to compile due to `TypeCheckFailure: Attribute node did not produce IR`.

```
struct User:
    balance:uint256

@external
def foo():
    a:User = User({balance:12})
    b:uint256 = a.balance
```

5.27 in and Not in Cannot Be Used With DynArray of Enums

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

CS-VYPER_MAY_2023-026

When trying to use the `in` or `not in` operator with a Dynamic Array of Enum, the compiler fails to compile the program with a `TypeMismatch`.

For example, the following contract does not compile due to the `in` operation.

```
enum A:
    a
    b
@external
def foo():
    f:DynArray[A,12] = []
    b:bool = A.a in f
```

6 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

6.1 Note on PR 3388

Note Version 1

The code generation of the constructor of a contract is performed before the code generation of the deployment version of its called functions. It hence relies on the fact that the code generation of runtime internal functions properly sets the frame information of the constructor's callees. If the runtime code generation would be to skip the generation of internal functions that will not be included in the runtime code for example, the `MemoryAllocator` of the constructor would be incorrectly initialized.

Additionally, following PR 3388, the following comment in the function `_runtime_ir` is now outdated:

```
# create a map of the IR functions since they might live in both
# runtime and deploy code (if init function calls them)
internal_functions_ir: list[IRnode] = []
```

6.2 Unused Parameters

Note Version 1

- `_is_function_implemented` does not use its parameter `fn_name`.
- `struct_literals` does not use its parameter `name`.