PUBLIC

Limited Code Review

of the Vyper Compiler Modules

February 28, 2024

Produced for

VYPER

by

CHAINSECURITY

Contents

1 Executive Summary 3

2 Review Overview 5

3 Limitations and use of report 10

4 Terminology 11

5 Findings 12

6 Informational 23

7 Notes 25

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 largest pull requests that were reviewed revamp the import system and introduce stateless and stateful modules to the Vyper language. The semantic analysis phase has been updated to support these new features and to be globally more robust. Constant folding has been modified so that it no longer breaks Vyper semantics. Additionally, more fined-grained variable read/write analyses have been introduced.

We find that the reviewed pull requests benefit both the language by adding new important features and the codebase in terms of consistency, readability and robustness. While the enforcement of type annotation for loop iterators improve greatly the type-checking phase, multiple issues related to loops were found as highlighted in Loop iterator overflow signed type, Double evaluation of range's start and Mistyped loop iterable.

Other important issues have that have been identified related to the layout override feature as shown in Overriding storage allocator does not handle stateful modules and Overriding storage allocator does not handle reentrant functions properly.

While no critical issues were found in the implementation of modules, we strongly recommend intensive testing of the new system before releasing it.

At the time of the review, the documentation of the modules system seems to be lacking and we recommend improving it.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. 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	1
Medium-Severity Findings	4
Low-Severity Findings	19

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 pull requests of the `https://github.com/vyperlang/vyper` repository:

The following pull requests were in scope:

- https://github.com/vyperlang/vyper/pull/3663
- https://github.com/vyperlang/vyper/pull/3729
- https://github.com/vyperlang/vyper/pull/3764
- https://github.com/vyperlang/vyper/pull/3769
- https://github.com/vyperlang/vyper/pull/3596
- https://github.com/vyperlang/vyper/pull/3719
- https://github.com/vyperlang/vyper/pull/3559
- https://github.com/vyperlang/vyper/pull/3679
- https://github.com/vyperlang/vyper/pull/3669
- https://github.com/vyperlang/vyper/pull/3655
- https://github.com/vyperlang/vyper/pull/3689
- https://github.com/vyperlang/vyper/pull/3697
- https://github.com/vyperlang/vyper/pull/3724
- https://github.com/vyperlang/vyper/pull/3731
- https://github.com/vyperlang/vyper/pull/3738
- https://github.com/vyperlang/vyper/pull/3762
- https://github.com/vyperlang/vyper/pull/3603

Due to the large amount of conflict between the different pull requests, the review was performed on the codebase as a whole on the commit `4b4e188ba83d28b5dd6ff66479e7448e5b925030`, rather than on the individual pull requests.

The following pull request was reviewed individually as it was not yet merged at the time of the review:

- https://github.com/vyperlang/vyper/pull/3817

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.

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

V	Date	Commit Hash	Note
1	14 February 2024	4b4e188ba83d28b5dd6ff66479e7448e5b925030	Initial Version

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. The semantics of the program are validated and the namespace is populated. The structure and the types of the program are checked and type annotations are added to the AST. Module imports are resolved and several related properties are checked.
4. Positions in storage and code are allocated for storage and immutable variables.
5. The Vyper AST is turned into a lower-level intermediate representation language (IR).
6. Various optimizations are applied to the IR.
7. The IR is turned into EVM assembly.
8. 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: semantic validation and code generation.

2.2.1 Semantic Validation

Once the Abstract Syntax Tree has been generated from the source code, 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. If an `Import` or an `ImportFrom` statement is visited, the imported module's AST is produced and analyzed by the `ModuleAnalyzer`.

The `FunctionAnalyzer` 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 perform some type-checking. Each statement's Sub-expressions are visited by the `ExprVisitor` which annotates the node with its type, and performs more semantic validation on the expression.

Once all module-level statements and functions have been properly analyzed, the compiler adds getters for public variables to the AST and checks some properties related to modules imported, such as ensuring that each initialized module's `_init` function is called or that used modules are used in the contract.

The `_ExprAnalyser` used when functions such `validate_expected_type` or `get_possible_types_from_node` are called as 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 `FunctionAnalyzer` 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. Module-level statements are also annotated with their type by the analyzer.

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 `ModuleT` containing the annotated AST as well as some properties such as the list of function definitions of the module or its variable declarations. 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. The compiler performs reachability analyses to avoid including unreachable code in the final bytecode.

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.

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	1

- Loop Iterator Overflow Signed Type

Medium-Severity Findings	4

- Double Evaluation of Range's Start
- Mistyped Loop Iterable
- Overriding Storage Allocator Does Not Handle Reentrant Functions Properly
- Overriding Storage Allocator Does Not Handle Stateful Modules

Low-Severity Findings	19

- Ambiguous Imports
- Assigning a Builtin to Itself Leads to Panic
- Codegen Fails for Raise and Assert With Non-Memory Messages
- Constants Cannot Be Imported From .vyi Files
- External Call Kwargs Allowed for Call to __init__
- Immutables Allowed to Be Used as Default Arguments
- Imprecise Duplicate Import Check
- Incorrect Error Messages
- Incorrect IR Identifier
- Initialization Analysis Can Have False Positive
- Misleading Error Message When Using @deploy on a Regular Function in an Interface
- Module Use Analysis Miss Nonreentrant Functions
- State Modifications Inside range Not Caught at Type-Checking
- Storage Variable of Type ModuleT and InterfaceT Not Prevented
- Un-precise Interface Body Check
- Variable Shadowing
- IfExp's body and orelse Children Can Have Arbitrary VyperType
- flag Can't Be Imported
- range Over Decimal Is Not Prevented

5.1 Loop Iterator Overflow Signed Type

Correctness High Version 1

CS-VYPER_FEBRUARY_2024-001

When a bound is specified, the `range()` built-in checks at runtime that `["assert", [le, start, end]]`. However, if the type of the loop iterator, is a signed integer, the check is not always correct as `le` here performs an unsigned integer comparison and not a signed one. This means that:

If the iterator type is `int256` it is possible to overflow 256 bits while looping if `end` is negative and `start` is positive.

For example, calling the function `foo` in the following contract returns `[57896044618658097711785492504343953926634992332820282019728792003956564819967, -57896044618658097711785492504343953926634992332820282019728792003956564819968, -57896044618658097711785492504343953926634992332820282019728792003956564819967]`

```
@external
def foo() -> DynArray[int256, 10]:
    res: DynArray[int256, 10] = empty(DynArray[int256, 10])
    x:int256 = max_value(int256)
    y:int256 = min_value(int256)+2
    for i:int256 in range(x,y , bound=10):
        res.append(i)
    return res
```

5.2 Double Evaluation of Range's Start

Correctness Medium Version 1

CS-VYPER_FEBRUARY_2024-002

The `start` of a range is evaluated twice, the result of the first evaluation is used as the actual starting value of the iteration. The second evaluation, which happens after the evaluation of the `end` argument, is used to ensure that `start<=end` and to compute the amount of `rounds` to be done.

Below is an example of this behavior. The starting value of `i` is `max_value(uint256)`, the amount of iterations to be done is `end-start = 3 - 1 = 2`. Calling the function returns hence `[115792089237316195423570985008687907853269984665640564039457584007913129639935, 0]`. This example further exploits this behavior to silently overflow the uint256 type with the iterator variable.

```
@external
def foo() -> DynArray[uint256, 3]:
    x:DynArray[uint256, 3] = [1,3,max_value(uint256)]
    res: DynArray[uint256, 3] = empty(DynArray[uint256, 3])
    for i:uint256 in range(x.pop(),x.pop(), bound = 3):
        res.append(i)

    return res
```

5.3 Mistyped Loop Iterable

Correctness Medium Version 1

CS-VYPER_FEBRUARY_2024-003

If a loop is defined over some non-literal static or dynamic array, the type obtained from the annotation of the iterator is not used to type-check the iterable. If the iterable's value type does not match the annotation, the type-checker will miss it and the compiler will later panic during code generation due to some assertion.

For example, the following code crashes the compiler with `vyper.exceptions.CodegenPanic: unhandled exception , parse_For`.

```
@external
@pure
def foo() :
    s:int256[3] = [1,-1,1]
    for i:uint256 in s:
        print(i)
```

5.4 Overriding Storage Allocator Does Not Handle Reentrant Functions Properly

Correctness **Medium** **Version 1**

CS-VYPER_FEBRUARY_2024-004

`OverridingStorageAllocator.set_storage_slots_with_overrides()` iterates over the `FunctionDef` AST nodes of the top-level module to find non-reentrant functions. If one is found, the compiler reserves some slot according to what was provided in the JSON file of storage slots overrides and annotates the function type with the given reentrant slot. However, for the next functions, the compiler will not annotate the function type with any slot. The issue is later caught at the code generation phase, where the compiler will crash.

For example, the following example would crash with `AttributeError: 'ContractFunctionT' object has no attribute 'reentrancy_key_position'`.

```
@nonreentrant
@external
def a():
    pass


@nonreentrant
@external
def b():
    pass
```

5.5 Overriding Storage Allocator Does Not Handle Stateful Modules

Correctness **Medium** **Version 1**

CS-VYPER_FEBRUARY_2024-005

`OverridingStorageAllocator.set_storage_slots_with_overrides()` does not handle stateful module imports and initialization properly and will not allocate storage and transient storage slots for variables or reentrant functions defined in a sub-module. The issue is only caught at the code generation stage and would most likely result in the compiler panicking.

5.6 Ambiguous Imports

Design | Low | Version 1

CS-VYPER_FEBRUARY_2024-006

If a module is named after one of the built-in interfaces Vyper provides and if that module is accessed via the path `ethereum.ercs`, then the module will be shadowed by the built-in interface. For example, if a module is named `ERC20` and is accessed via `from ethereum.ercs import ERC20`, the Vyper builtin `ERC20` will be imported instead.

Similarly, if a module `vyper.interfaces.ERC20` is defined and imported, the compiler would fail with `vyper.exceptions.ModuleNotFound: vyper.interfaces.ERC20 (hint: try renaming ` vyper.interfaces` to `ethereum.ercs`)`.

5.7 Assigning a Builtin to Itself Leads to Panic

Design | Low | Version 1

CS-VYPER_FEBRUARY_2024-007

Assigning some builtin to itself is not prevented and leads to a compiler panic.

For example, compiling the following code leads the compiler to raise the following exception: `CodegenPanic: unhandled exception , parse_Name`.

```
@external
def foo():
    convert = convert
```

5.8 Codegen Fails for `Raise` and `Assert` With Non-Memory Messages

Correctness | Low | Version 1

CS-VYPER_FEBRUARY_2024-008

The code generation for `Raise` and `Assert` fails if the reason string is not a memory variable. This behavior is due to `Stmt._assert_reason` calling `make_byte_array_copier` with `buf` instead of an IR node whose value is `buf` and whose location is `memory` as it would be done by `ensure_in_memory` for example.

For example, compiling the following contract leads to the compiler panicking with `vyper.exceptions.CodegenPanic:`
`unhandled exception 'int' object has no attribute 'typ', parse_Raise`.

```
a:String[1]
@external
def foo():
    self.a = "a"
    raise self.a
```

5.9 Constants Cannot Be Imported From `.vyi` Files

Design Low Version 1

CS-VYPER_FEBRUARY_2024-009

Constants cannot be imported from `.vyi` interface files. This behavior would be useful as one would usually want this for types dependent on some value (for example a static array with some length).

5.10 External Call Kwargs Allowed for Call to `__init__`

Correctness Low Version 1

CS-VYPER_FEBRUARY_2024-010

`ContractFunctionT.fetch_call_return` depends on `self.is_internal` to prevent or not the call from being passed some external call reserved kwargs such as `gas` or `value`. However, an `__init__` function has `self.is_internal = False` so it is possible to pass kwags when calling it although it should not be allowed.

For example, the contract below compiles:

```
# main.vy
import bar

initializes: bar

@deploy
@payable
def __init__():
    bar.__init__(12, gas=14, value = 12, skip_contract_check= True)
```

```
# bar.vy
a:uint256

@deploy
@payable
def __init__(x: uint256):
    self.a = x
```

5.11 Immutables Allowed to Be Used as Default Arguments

Design Low Version 1

According to the documentation, the default arguments of a function must be literals or environment variables, however, immutables are allowed to be used as default arguments.

The following contract compiles:

```
x:immutable(uint256)

@deploy
def __init__():
    x = 1

@external
def foo(val:uint256 = x):
    pass
```

5.12 Imprecise Duplicate Import Check

Design | Low | Version 1

CS-VYPER_FEBRUARY_2024-012

In `ModuleAnalyzer._load_import_helper()`, the following check ensures that a given file is not imported multiple times:

```
if path in self._imported_modules:
    previous_import_stmt = self._imported_modules[path]
    raise DuplicateImport(f"{alias} imported more than once!", previous_import_stmt, node)
```

However, the check is not performed using a normalized path as it is being done in the input bundle. This means that for a `lib1` in the directory `project`, it can be bypassed as follows:

```
import lib1
from ..project import lib1 as lib2
```

Note however that since paths are normalized in the input bundle, if the two modules are initialized, for example, the compiler will raise an exception.

5.13 Incorrect Error Messages

Correctness | Low | Version 1

CS-VYPER_FEBRUARY_2024-013

- In `VariableDecl.validate()`, instead of calling the method `self._pretty_location()` to pretty-print the location of the given variable, `self._pretty_location` is used.
- In `_CreateBase.build_IR`, `context.check_is_not_constant` should be called with an f-string and not a regular string to have `self._id` as the id of the built-in in the error message.

5.14 Incorrect IR Identifier

Correctness | Low | Version 1

CS-VYPER_FEBRUARY_2024-014

The new `deploy` visibility is used to abstract away the fact that the constructor can now behave either as an actual constructor or as internal functions callable in the deployment context. However, `_FuncIRInfo.visibility()` has not been updated with this new semantics, leading `_FuncIRInfo.ir_identifier()` to return some identifier marked as `external` although the constructor might be treated as an internal function.

5.15 Initialization Analysis Can Have False Positive

Design Low Version 1

CS-VYPER_FEBRUARY_2024-015

When analyzing that the constructor of an imported module is called in the constructor of a module that imports and initializes it, `ModuleAnalyzer.validate_initialized_modules()` only checks that an `__init__` call for the imported module appears once and only once in the AST of the importing module's constructor. For similar reasons as the PR3162 issue, this is not enough as the call might appear in a loop or in an `if else` statement. It could hence be possible either to have some call path that never calls the constructor of the imported module or to have it called multiple times.

For example, given some module a with an `__init__` function, the following code would compile without any error although the imported module's `__init__` function is either called twice or never called depending on the value of b.

```
import a
initializes: a

@deploy
def __init__(b:bool):
    if b:
        for i:uint256 in range(2):
            a.__init__()
    else:
        pass
```

5.16 Misleading Error Message When Using `@deploy` on a Regular Function in an Interface

Design Low Version 1

CS-VYPER_FEBRUARY_2024-016

In `ContractFunctionT.from_FunctionDef()`, the following check is performed:

```
if function_visibility == FunctionVisibility.DEPLOY and funcdef.name != "__init__":
raise FunctionDeclarationException(
    "Only constructors can be marked as `@deploy`!", funcdef
)
```

Having this check moved to `_parsed_decorators` which is called by both `ContractFunctionT.from_FunctionDef()` and `ContractFunctionT.from_vyi()` would allow performing the same check for imported interfaces. Currently, if a regular function in an interface is decorated as `@deploy`, the compiler will fail with a misleading error message `` Internal functions in `.vyi` files are not allowed!``?

5.17 Module Use Analysis Miss Nonreentrant Functions

Correctness Low Version 1

CS-VYPER_FEBRUARY_2024-017

When a module is never initialized, it should only be allowed to call functions that are stateless in the sense that they do not access storage or immutables. However, the analysis does not account for non-reentrant decorators. If a non-initialized module's non-reentrant function is called, the compiler will crash during code generation after successfully passing semantic analysis.

Compiling the example below crashes the compiler with `AttributeError: 'ContractFunctionT' object has no attribute 'reentrancy_key_position'`.

```
# main.vy
import lib

@external
def foo():
    lib.bar()
```

```
# lib.vy
@nonreentrant
@internal
def bar():
    pass
```

5.18 State Modifications Inside `range` Not Caught at Type-Checking

Design Low Version 1

CS-VYPER_FEBRUARY_2024-018

Sub-expressions of a range expression should have a `constant` constancy, in other words, they should not be able to update the storage. However, this is not enforced during type-checking and is only caught by the following assertion in `ir_for_self_call()`:

```
# CMC 2023-05-17 this seems like it is already caught in typechecker
if context.is_constant() and func_t.is_mutable:
    raise StateAccessViolation(
        f"May not call state modifying function "
        f"'{method_name}' within {context.pp_constancy()}.",
        stmt_expr,
    )
```

This is the case of the example below:

```
x:uint256
```

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

@external
def foo():
    for i:uint256 in range(self.bar(), bound=10):
        pass
```

5.19 Storage Variable of Type `ModuleT` and `InterfaceT` Not Prevented

Design Low Version 1

CS-VYPER_FEBRUARY_2024-019

Since `ModuleT` is a Vyper type, and since `ModuleT._invalid_locations` does not contain `DataLocation.STORAGE`, it is possible to use it as a type for a storage variable while this should not be possible.

The contract below will compile, and calling `foo()` will return `(1,2)`.

```
# main.vy
import lib

initializes: lib

x:lib
y:lib

@external
def foo() -> (uint256, uint256):
    return (self.x.bar(), self.y.bar())
```

```
# lib.vy
a:uint256

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

Note that a similar behavior can be observed for `InterfaceT`.

5.20 Un-precise Interface Body Check

Design Low Version 1

CS-VYPER_FEBRUARY_2024-020

In interface files (`.vyi`), function bodies should only contain the AST Ellipsis node (`...`). However given how the corresponding check is done, any statement node with a `value` field containing an Ellipsis node can be used in place of the `Expr` AST node.

```
if len(funcdef.body) != 1 or not isinstance(funcdef.body[0].get("value"), vy_ast.Ellipsis):
    raise FunctionDeclarationException(
        "function body in an interface can only be `...`!", funcdef
    )
```

For example, compiling a contract importing the following interface does not raise any exception:

```
@external
def foo():
    log ...
```

5.21 Variable Shadowing

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

CS-VYPER_FEBRUARY_2024-021

In `InterfaceT._from_lists()`, the argument name is shadowed by the different loop iterators. This leads some function, event, or struct name to be used as the name of the `InterfaceT` to be constructed.

5.22 `IfExp`'s `body` and `orelse` Children Can Have Arbitrary `VyperType`

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

CS-VYPER_FEBRUARY_2024-022

The `ExprVisitor` does not enforce any type on the `IfExp`'s `body` and `orelse` children except that they should have the same type. This means that using arbitrary `VyperType` subclasses like `ModuleT`, `BuiltinFunctionT` or `SelfT` is not prevented and either results in a successful compilation or the compiler panicking during the code generation phase.

Note that this issue relates to Issue_3513

For example, the following contract compiles successfully:

```
# main.vy
import lib1

initializes: lib1

@external
def foo():
    (lib1 if True else lib1).bar(1)
```

```
# lib1.vy

@internal
```

```
def bar(x: uint256):
    pass
```

The contracts below would fail to compile respectively with:

- `vyper.exceptions.CodegenPanic: unhandled exception None, parse_Attribute`
- `vyper.exceptions.CompilerPanic: Unreachable`

```
@external
def foo():
    x:uint256 = (self if True else self).balance
```

```
@external
def foo():
    x:uint256 = (min if True else min)(1,2)
```

5.23 `flag` Can't Be Imported

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

CS-VYPER_FEBRUARY_2024-023

While it is possible to import events and struct types both from `.vy` and `.vyi` files, it is not possible to import flags.

For interfaces, this is because `InterfaceT` does not hold the flags defined in the interface and hence `InterfaceT.get_type_member()` only returns the set of events and struct types defined.

For modules, similarly, flag types are not added to the set of members of the `ModuleT` at construction time.

5.24 `range` Over Decimal Is Not Prevented

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

CS-VYPER_FEBRUARY_2024-024

The `range()` built-in function should only accept integer types. However, this is not enforced and instead, only the AST type of the arguments of the range is checked: either both `start` and `stop` should be `Num` nodes, or `bound` should be a `Num` node. This means that having them as decimal numbers is not prevented since Vyper AST type `Decimal` is a subclass of `Num`.

For example, the following code is valid:

```
@external
@pure
def foo():
    end: decimal = 2.2
    for i: decimal in range(1.1, end, bound=10.1):
        pass
```

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 Inconsistent Documentation

Informational **Version 1**

CS-VYPER_FEBRUARY_2024-025

The function `parse_to_ast_with_settings` is documented as taking a `contract_name` argument, but it does not. Additionally, the documentation mentions twice the `source_id` argument.

6.2 Misleading Function Name

Informational **Version 1**

CS-VYPER_FEBRUARY_2024-026

The function `_parse_and_fold_ast` is misleading as it does not perform any folding on the AST.

6.3 Range's `bound` Typing

Informational **Version 1**

CS-VYPER_FEBRUARY_2024-027

When the compiler type-checks a `range` expression, it ensures that the `start`, `stop`, and `bound` expressions all have the same type and that this type matches the one provided in the annotation of the iterator. However, it might be that the `bound` is legitimately greater than the maximum value of the given type, in such case the compiler would raise an error.

For example, the following two programs should behave similarly, however, the first one would compile successfully and one could call `foo()` without any revert while the second one would not compile as `137` is not an `int8`.

```
@external
def foo():
    # the loop performs 137 iterations
    for i:int8 in range(-10, max_value(int8)):
        pass
```

```
@external
def foo():
    end:int8 = max_value(int8)
    for i:int8 in range(-10, end, bound = 137):
        pass
```

6.4 Redundant Check of Module Use

Informational **Version 1**

CS-VYPER_FEBRUARY_2024-028

In `ExprVisitor.visit_Call()`, the following is done:

```
if self.function_analyzer:
self._check_call_mutability(func_type.mutability)

for s in func_type.get_variable_accesses():
    if s.variable.is_module_variable():
        self.function_analyzer._check_module_use(node.func)

if func_type.is_deploy and not self.func.is_deploy:
    raise CallViolation(
        f"Cannot call an @{func_type.visibility} function from "
        f"an @{self.func.visibility} function!",
        node,
    )
```

The call to `_check_module_use()` is done as many times as there are variable accesses in the called function, which is redundant as it does not depend on the variable accesses and could be done only once.

6.5 `ModuleT` Cached Properties Dependent on Properties

Informational **Version 1**

CS-VYPER_FEBRUARY_2024-029

In `ModuleT`, several cached properties such as `variables`, `functions` or `immutables` depend on some non-cached properties. This means that if a cached property is to be used before the value returned by the correspondent property is final, the compiler will always use this out-of-date cached value. Although no example of this behavior was found, such a pattern can lead to issues if, in the future, some of these properties were to be used before the corresponding cached property is fully set. An example would be if `functions` were to be read before the AST expansion which updates `function_defs`.

6.6 `enum` Not Renamed to `flag`

Informational **Version 1**

CS-VYPER_FEBRUARY_2024-030

PR3697 renames the `enum` keyword to `flag` however not all occurrences of `enum` are renamed in the code, comments and documentation.

For example `FlagT._enum_members` should be renamed to `FlagT._flag_members`.

7 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.

7.1 Ambiguous Imports Are Allowed

Note **Version 1**

In `InputBundle.load_file()`, once a valid path is found, the method returns even though there might be multiple matching paths. It could be beneficial to raise an exception in case there are multiple paths to avoid any ambiguous import.