PUBLIC

Limited Code Review

of the Vyper Compiler

Code generation

April 26, 2024

Produced for

VYPER

by

CHAINSECURITY

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 two weeks. Given the large scope and codebase and the limited time, the findings aren't exhaustive.

This review concentrated on the code generation phase of the Vyper compiler. Since no major new features have been added to the code generation recently, the review primarily examined the existing codebase along with recent fixes.

The most critical subjects covered in our review are the order of evaluation, double evaluation and bound checking.

Both orders of evaluation and double evaluation have been recurrent issues in the Vyper compiler, the two issues Multiple evaluations of the target of byte and dynamic array assignments and Incorrect order of evaluation are examples of this and extend the previously known issues in these areas.

Bound checking also appears to be a critical subject recently as shown in the issue Looping over a negative range always reverts and multiple issues from past reviews.

General subjects covered in our review include constant folding, highlighted in Folding not supported in certain locations, optimizations soundness shown in Missing continue in assembly optimizations and correctness of the memory accesses as shown in multiple issues.

It is important to note that security reviews are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

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	0
Medium-Severity Findings	4
Low-Severity Findings	16

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 source code files inside the Vyper Compiler repository:

The review consisted of a general review of `vyper.codegen` and `vyper.ir` together with the review of specific pull requests and was conducted within the available time constraints.

The following pull requests were in scope:

- https://github.com/vyperlang/vyper/pull/3818
- https://github.com/vyperlang/vyper/pull/3874
- https://github.com/vyperlang/vyper/pull/3844
- https://github.com/vyperlang/vyper/pull/3814

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

- https://github.com/vyperlang/vyper/pull/3924

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 Vyper Compiler 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	April 09, 2024	b43fface148b45cbd2c90b5d24f77398a63745b5	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 and analysis 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 modules annotated as used are actually used in the contract.

The `_ExprAnalyser` is used when functions such `validate_expected_type` or `get_possible_types_from_node` are called 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 or 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. This is possible because there are no dynamic types in Vyper and all variable size is known at compile time. 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 code size 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	0

Medium-Severity Findings	4

- Arbitrary Memory Read in _unpack_returndata()
- Incorrect Order of Evaluation
- Looping Over a Negative Range Always Reverts
- Multiple Evaluations of the Target of Byte and Dynamic Array Assignments

Low-Severity Findings	16

- Assembly Optimization's Condition Could Be Restricted
- Compiler Panic When Popping From or Appending to an Empty Array
- F-Strings Missing Prefix
- Folded Nodes Are Not Validated
- Folding Not Supported in Certain Locations
- IR Node Annotation Is Always Overwritten With AST Node Source Code Annotation
- Incorrect Buffer Size Passed to abi_encode()
- Incorrect Documentation for shift()
- Incorrect IR Node Annotation
- Indexing Empty Arrays Not Prevented
- Missing Constancy Check for Transient Storage
- Missing Continue in Assembly Optimizations
- Non-constant Nodes Can Be Folded
- _handle_modification Fails for Call Nodes
- _handle_modification Fails for IfExp Nodes
- make_byte_array_copier() Copies Unnecessary Words

5.1 Arbitrary Memory Read in `_unpack_returndata()`

Design Medium Version 1

CS-VYPER_APRIL_2024-001

The known `_abi_decode` memory overflow (GHSA-9p8r-4xp4-gw5w) allows for reading arbitrary memory using a malicious payload instead of some properly ABI encoded data. However, this issue is also present when unpacking the return data of an external call (`_unpack_returndata()`)

```
@external
def bar() -> (uint256, uint256, uint256):
    return (192, 0, 0)


interface A:
    def bar() -> String[32]: nonpayable

@internal
def internal_func() -> String[32]:
    # in this case we extcall ourself for the sake of the POC but it could be any other contract
    x:String[32] = extcall A(self).bar()
    return x
@external
def foo() -> String[32]:
    w:String[8] = "oooops"
    return self.internal_func() # return "oooops"
```

5.2 Incorrect Order of Evaluation

Design Medium Version 1

CS-VYPER_APRIL_2024-002

The order of evaluation in Vyper is specified as left-to-right; in the same order as the expressions appear in the the source code.

- Due to the way IR nodes that match directly EVM opcodes are converted to assembly, for some built-ins and operators, it is known that it might not be the case as described in GHSA-g2xh-c426-v8mf.
- The compiler uses a pattern `cache_when_complex` to cache the result of a complex expression and avoid double evaluation. However, this pattern appears to be problematic regarding the order of evaluation.
    - Given an IR node that has multiple children that should be evaluated in a specific order, it might be that not all children are cached. A cached complex child would hence be evaluated before any non-cached children independently of their order in the source code. This behavior is notably seen in assignments given that `make_setter()` caches `src` but not `dst`.
    - A more subtle issue is that the compiler might cache all children of an IR node, however, one or multiple nodes are not considered as complex and their evaluation is inlined. although in that case, a non-complex node cannot have side effects, its evaluation can still read (e.g a `SLOAD`) side effects from a complex node that was cached and evaluated first. This case appears to be problematic for the `slice()` and `extract32()` built-ins.
- Another issue regarding the order of evaluation is that the order of evaluation of the kwargs passed to built-ins and call expressions do not follow the order of the source code.

The following examples show multiple cases where the order of evaluation is right-to-left.

```
i:uint256

@internal
def change_i() -> uint256:
    self.i += 1
    return 12

@external
def foo() -> DynArray[uint256,2]:
    x:DynArray[uint256,2] = [1,2]
    x[self.i] += self.change_i()
    return x # returns [13,2]
```

```
boo:Bytes[32]

@internal
def bar() -> uint256:
    self.boo = b'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'
    return 0

@internal
def baz() -> Bytes[32]:
    return self.boo

@external
def slice() -> (Bytes[32], Bytes[32]):
    self.boo = b'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb'
    s: Bytes[1] = slice(self.boo, self.bar(), 1)

    self.boo = b'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb'
    t: Bytes[1] = slice(self.baz(), self.bar(), 1)

    return s,t # (b'a', b'b')

@external
def extract32() -> (bytes32, bytes32):
    self.boo = b'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb'
    s: bytes32 = extract32(self.boo, self.bar())

    self.boo = b'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb'
    t: bytes32 = extract32(self.baz(), self.bar())

    return s,t #(b'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa', b'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb')
```

```
@external
@payable
def bar(): return

interface Bar:
    def bar(): payable

x: uint256

@internal
def gas() -> uint256:
    self.x = 2
    return 100000

@internal
def value() -> uint256:
    self.x = 1
    return 0
```

```
@external
@payable
def foo():
    extcall Bar(self).bar(gas=self.gas(), value=self.value())
    temp: uint256 = self.x
    extcall Bar(self).bar(value=self.value(), gas=self.gas())
    assert self.x == temp # passes
```

5.3 Looping Over a Negative Range Always Reverts

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

CS-VYPER_APRIL_2024-003

When looping over a range of the form `range(start, start + N)`, if `start` is negative, the execution will always revert.

This issue is caused by an incorrect assertion inserted by the code generation of the range (`stmt.parse_For_range()`):

```
_, hi = start.typ.int_bounds
start = clamp("le", start, hi + 1 - rounds)
```

This assertion was introduced in commit 3de1415ee7 to fix GHSA-6r8q-pfpv-7cgj. The issue arises when `start` is signed, instead of using `sle`, `le` is used and `start` is interpreted as an unsigned integer for the comparison. If it is a negative number, its 255th bit is set to `1` and is hence interpreted as a very large unsigned integer making the assertion always fail.

Note that this issue has been fixed in PR 3679 and was no longer present on the commit used for this review.

```
@external
def foo():
    x:int256 = min_value(int256)
    # revert when it should not since we have the following assertion that fails:
    # [assert, [le, min_value(int256), max_value(int256) + 1 - 10]],
    for i in range(x, x + 10):
        pass
```

5.4 Multiple Evaluations of the Target of Byte and Dynamic Array Assignments

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

CS-VYPER_APRIL_2024-004

Multiple evaluations of the target of byte and dynamic array assignments have been described in issue 3514. However, this issue was incomplete since it would only show the case where the complex expression being evaluated twice is caught by the compiler's "non-unique symbol" sanity check. However, it is possible to have a complex expression evaluated twice without the compiler crashing by using builtins like `raw_call` or the `create_` class or `.pop()` on a non-storage array.

Note that this issue has been partially mitigated by 3835 given that builtins like `raw_call` and the `create_` class are now emitting unique symbols, however, this is not yet the case for `.pop()` when the array is not in storage.

The following code will compile successfully, and `x.pop()` will be performed twice:

```
d:DynArray[DynArray[uint256,1],1]
addr:address
@external
def foo() -> DynArray[uint256, 2]:
    self.d = [[0]]
    x:DynArray[uint256, 2] = [0,0]
    self.d[x.pop()] = [1]
    return x # return [] instead of [0]
```

5.5 Assembly Optimization's Condition Could Be Restricted

Design Low Version 1

CS-VYPER_APRIL_2024-005

`_stack_peephole_opts()` try to perform several optimizations as long as `i < len(assembly) - 2`, however, the last two optimizations could, in theory, be performed even if `i == len(assembly) - 1` given that they only access `assembly[i]` and `assembly[i+1]`.

```
if (
    isinstance(assembly[i], str)
    and assembly[i].startswith("SWAP")
    and assembly[i] == assembly[i + 1]
):
    changed = True
    del assembly[i : i + 2]
if assembly[i] == "SWAP1" and assembly[i + 1].lower() in COMMUTATIVE_OPS:
    changed = True
    del assembly[i]
```

5.6 Compiler Panic When Popping From or Appending to an Empty Array

Correctness Low Version 1

CS-VYPER_APRIL_2024-006

When pop-ing from or append-ing to an empty array (`empty(T)`), the compiler panics when trying to call `FunctionAnalyzer._handle_modification()` on the empty array.

The following code will make the compiler panic:

```
@external
def bar():
    a:uint128 = empty(DynArray[uint128,2]).pop() # CompilerPanic: unhandled exception
```

```
@external
def bar():
    empty(DynArray[uint128,2]).append(1) # CompilerPanic: unhandled exception
```

5.7 F-Strings Missing Prefix

Correctness Low Version 1

CS-VYPER_APRIL_2024-007

The following f-strings are missing the `f` prefix:

- `vyper/ast/nodes.py:1394:41`
- `vyper/ast/pre_parser.py:207:54`
- `vyper/builtins/functions.py:1662:39`
- `vyper/builtins/functions.py:2280:37`
- `vyper/cli/vyper_compile.py:286:17`
- `vyper/cli/vyper_compile.py:287:17`
- `vyper/cli/vyper_json.py:193:29`
- `vyper/cli/vyper_json.py:202:17`
- `vyper/cli/vyper_json.py:206:17`
- `vyper/codegen/arithmetic.py:292:42`
- `vyper/codegen/core.py:602:32`
- `vyper/semantics/analysis/base.py:283:41`
- `vyper/semantics/analysis/utils.py:358:41`
- `vyper/semantics/namespace.py:96:24`
- `vyper/semantics/types/base.py:172:29`
- `vyper/semantics/types/primitives.py:97:34`
- `vyper/semantics/types/utils.py:163:29`

5.8 Folded Nodes Are Not Validated

Design Low Version 1

CS-VYPER_APRIL_2024-008

When some node is folded the folded value is not validated. In the general case, it is not an issue as the semantic validation phase will catch the error. However, if the folded value is used in a type annotation for a `String` or `Byte` type, the compiler will not raise.

The following contract compiles although the size in byte of the String is greater than the maximum value of uint256

```
x:String[max_value(uint256) + 10]
```

5.9 Folding Not Supported in Certain Locations

Design Low Version 1

CS-VYPER_APRIL_2024-009

Several Vyper constructs require or can be optimized if constant values are provided. To achieve this, AST nodes can be folded to constants by the compiler, however, the following locations do not support folding:

- `length` in `slice(x, start, length)` when x is `msg.data` or `address.code`:
    - `isinstance(parent.args[2], vy_ast.Int)` in `_validate_msg_data_attribute()`
    - `isinstance(parent.args[2], vy_ast.Int)` in `_validate_address_code()`
- `length` in `slice(x, start, length)`:
    - `if length_literal is not None` condition in `Slice.fetch_call_return()`
- `index` in `variable[index]` when the variable is an array.
    - `isinstance(node, vy_ast.Int)` in `_SequenceT.validate_index_type()`
- `index` in `variable[index]` when the variable is a tuple.
    - `not isinstance(node, vy_ast.Int)` in `TupleT.validate_index_type()`
- x or y in `x ** y`:
    - `left, right = _get_lr()` in `NumericT.validate_numeric_op()`
- x in `convert(x, T)`
    - `_convert._to_int()`, `_convert.to_decimal()` and `_convert._cast_bytestring()` does not check if `expr` has a folded value.
- `topic` in `raw_log(topic, data)` .
    - `not isinstance(node.args[0], vy_ast.List)` in `RawLog.infer_arg_types()`

The following example demonstrates the different issues.

```
@external
def foo():
    x: Bytes[32] = slice(msg.data, 0, 31 + 1) # StructureException
```

```
@external
def foo(tuple: (uint256,uint256)) -> uint256:
    return tuple[0+1] # InvalidType
```

```
k: constant(Bytes[3]) = b'aaa'
@external
def foo():

    a: Bytes[2] = convert (k, Bytes[2]) # compiles instead of failing
    b: Bytes[2] = convert (b'aaa', Bytes[2]) # TypeMismatch
```

```
a: constant(uint256) = 12
@external
def foo():
    # The following check is inserted by the compiler although it is not necessary
    # [assert, [iszero, [shr, 128, 12 <12>]]],
    b: uint128 = convert (a, uint128)
```

```
topic: constant(bytes32) = 0x1212121212121210212801291212121212121210121212121212121212121212
@external
def foo():
    raw_log([[topic]][0], b'') # InvalidType
```

5.10 IR Node Annotation Is Always Overwritten With AST Node Source Code Annotation

Design Low Version 1

CS-VYPER_APRIL_2024-010

In `Expr.__init__()` and `Stmt.__init__()`, the IR node annotation is always overwritten with the AST node source code annotation. In some cases, it might be that `self.ir_node.annotation` is already set to a meaningful annotation and that the AST node source code annotation is empty or less meaningful.

5.11 Incorrect Buffer Size Passed to `abi_encode()`

Correctness Low Version 1

CS-VYPER_APRIL_2024-011

In `external_call.py` the function `_pack_arguments` encodes the arguments of the call as follow:

```
if len(args) != 0:
    pack_args.append(abi_encode(buf + 32, args_as_tuple, context, bufsz=buflen))
```

However, the buffer size passed is incorrect given that `buflen` is the length of the buffer allocated at `buf` but here the passed buffer starts at `buf + 32`.

No security implications were found for this issue.

5.12 Incorrect Documentation for `shift()`

Correctness Low Version 1

CS-VYPER_APRIL_2024-012

According to the documentation, the now deprecated `shift` built-in is supposed to return values of type `uint256`. However, when called on a value of type `int256` it returns an `int256`.

5.13 Incorrect IR Node Annotation

Design Low Version 1

CS-VYPER_APRIL_2024-013

The following creates an IR node for global attribute `expr.parse_Attribute()`. However, with the introduction of modules, it is no longer safe to assume that the attribute is from the `self` object.

```
if (varinfo := self.expr._expr_info.var_info) is not None:
    if varinfo.is_constant:
        return Expr.parse_value_expr(varinfo.decl_node.value, self.context)

    location = data_location_to_address_space(
        varinfo.location, self.context.is_ctor_context
    )

    ret = IRnode.from_list(
        varinfo.position.position,
        typ=varinfo.typ,
        location=location,
        annotation="self." + self.expr.attr,
    )
    ret._referenced_variables = {varinfo}

    return ret
```

5.14 Indexing Empty Arrays Not Prevented

Design Low Version 1

CS-VYPER_APRIL_2024-014

Indexing an empty array is not caught by the type checker and causes the code generation to fail with a compiler panic.

Compiling the following contract will cause a compiler panic:

```
@external
def bar(x:uint128):
    a:uint128 = empty(DynArray[uint128,2])[0]
```

```
vyper.exceptions.TypeCheckFailure: indexing into zero array not allowed
```

5.15 Missing Constancy Check for Transient Storage

Correctness Low Version 1

CS-VYPER_APRIL_2024-015

In `Stmt._get_target()`, the target is checked not to have a storage location in the case the context is constant. However, the check is missing for the case where the target is in transient storage. Given that the semantic analysis phase should make the compiler fail earlier if transient storage is accessed in a constant context, no impact of the missing check could be found.

5.16 Missing Continue in Assembly Optimizations

Design Low Version 1

CS-VYPER_APRIL_2024-016

`_stack_peephole_opts()` tries to perform 5 different optimizations at each index `i` of the assembly using a while loop. However, the last two optimizations, when performed do not `continue` the loop, which means that:

- `i` is incremented by `1`, leading to skipping the next instruction and potentially missing an optimization.
- If both optimizations are performed for a given `i`, it might be that the second one tries accessing an index that is out of range given that the first one removed the 2 last elements of the array and the check for the index is not performed in between the two optimizations.

5.17 Non-constant Nodes Can Be Folded

Correctness Low Version 1

CS-VYPER_APRIL_2024-017

`Subscript.evaluate()` does not check if all elements of the array are constant before folding the expression. This can lead to incorrectly folding the expression since non-constant nodes are removed from the AST and not evaluated. Note that the issue is related to issue 3442 and is no longer present in the commit used for this review given that it was fixed by PR 3669.

```
c: uint256

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

@external
def foo() -> uint256:
    l: uint256 = [self.bar(), self.bar()][0]
    return self.c # returns 1
```

5.18 `_handle_modification` Fails for `Call` Nodes

Correctness Low Version 1

CS-VYPER_APRIL_2024-018

When an `Call` node is passed to `_handle_modification`, the function fails to handle it and the compiler panics with an unhandled exception.

The following example demonstrates this behavior:

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

@external
def foo():
    x: uint256 = self.c.pop().pop() # CompilerPanic: unhandled exception
```

5.19 `_handle_modification` Fails for `IfExp` Nodes

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

CS-VYPER_APRIL_2024-019

When an `IfExp` node is passed to `_handle_modification`, the function fails to handle it and the compiler panics with an unhandled exception.

The following examples demonstrate this behavior:

```
@external
def bar():
    d: DynArray[uint256, 2] = [1,2]
    (d if True else d)[1] = 1 # CompilerPanic: unhandled exception
```

```
@external
def bar():
    d: DynArray[uint256, 2] = [1]
    x:uint256 = (d if True else d).pop() # CompilerPanic: unhandled exception
```

```
@external
def bar():
    d: DynArray[uint256, 2] = [1]
    (d if True else d).append(1) # CompilerPanic: unhandled exception
```

5.20 `make_byte_array_copier()` Copies Unnecessary Words

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

CS-VYPER_APRIL_2024-020

In certain conditions, `make_byte_array_copier()` uses a special path to copy the data:

```
with src.cache_when_complex("src") as (b1, src):
    has_storage = STORAGE in (src.location, dst.location)
    is_memory_copy = dst.location == src.location == MEMORY
    batch_uses_identity = is_memory_copy and not version_check(begin="cancun")
    if src.typ.maxlen <= 32 and (has_storage or batch_uses_identity):
        # it's cheaper to run two load/stores instead of copy_bytes

        ret = ["seq"]
        # store length word
        len_ = get_bytearray_length(src)
        ret.append(STORE(dst, len_))

        # store the single data word.
        dst_data_ptr = bytes_data_ptr(dst)
        src_data_ptr = bytes_data_ptr(src)
        ret.append(STORE(dst_data_ptr, LOAD(src_data_ptr)))
        return b1.resolve(ret)
```

When the source's dynamic length is 0, `STORE` and `LOAD` operations are performed although they are not necessary and somewhat access out-of-bounds dirty data.

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 Argument and Return Buffer Size Overestimated for External Calls

Informational **Version 1**

CS-VYPER_APRIL_2024-021

In `external_call.py` the function `_pack_arguments` computes the length of the buffer used to store both the arguments of the call and its returned value as:

```
if fn_type.return_type is not None:
    return_abi_t = calculate_type_for_external_return(fn_type.return_type).abi_type

    # we use the same buffer for args and returndata,
    # so allocate enough space here for the returndata too.
    buflen = max(args_abi_t.size_bound(), return_abi_t.size_bound())
else:
    buflen = args_abi_t.size_bound()

buflen += 32  # padding for the method id
```

In case the function returns a value and `return_abi_t.size_bound()` is greater than or equal to `args_abi_t.size_bound() + 32`, the last 32 bytes of the buffer are never used.

6.2 Duplicated Code

Informational **Version 1**

CS-VYPER_APRIL_2024-022

The following is performed in `ir_for_self_call()`:

```
goto_op = ["goto", func_t._ir_info.internal_function_label(context.is_ctor_context)]
```

However, `_label` is set to `func_t._ir_info.internal_function_label(context.is_ctor_context)` earlier in the function and could be used to avoid duplicated code.

6.3 Exception Raised During Optimization Are Not `VyperInternalException`

Informational **Version 1**

CS-VYPER_APRIL_2024-023

Exceptions raised as part of the code generation (`module.generate_ir_for_module()`) are wrapped in a `CodegenError` exception for easier debugging. This is not the case for exceptions raised during the optimization (`optimizer.optimize()`).

For example, the following contract fails to compile with an `AssertionError` and not a Vyper exception:

```
c: uint256

@internal
def se_bbytes() -> Bytes[4]:
    self.c += 1
    return b''

@external
def foo(a: address) -> uint256:
    x:address = create_from_blueprint(a, self.se_bbytes(), code_offset=1, raw_args = True)
    return self.c
```

6.4 Imprecise Assertion in `Expr.handle_binop()`

Informational **Version 1**

CS-VYPER_APRIL_2024-024

In `Expr.handle_binop()`, the assertion `assert is_numeric_type(left.typ)` is performed after the handling of `vy_ast.LShift` and `vy_ast.RShift`, which are only defined for numeric types. The assertion could be moved before the handling of these operators to be more precise.

6.5 Imprecise Sanity Check

Informational **Version 1**

CS-VYPER_APRIL_2024-025

In `_check_assign_list()`, the following code is used to handle static array assignment:

```
if isinstance(left.typ, SArrayT):
    if not is_array_like(right.typ):  # pragma: nocover
        FAIL()
    if left.typ.count != right.typ.count:  # pragma: nocover
        FAIL()

    # TODO recurse into left, right if literals?
    check_assign(
        dummy_node_for_type(left.typ.value_type), dummy_node_for_type(right.typ.value_type)
    )
```

the check `if not is_array_like(right.typ)` could be more precise since dynamic arrays cannot be assigned to static arrays.

6.6 Incomplete Error Message

Informational **Version 1**

CS-VYPER_APRIL_2024-026

The following error message in `HashMapT.from_annotation()` is unclear as Bytes array and String can also be used as key types of HashMaps.

```
if not key_type._as_hashmap_key:
    raise InvalidType("can only use primitive types as HashMap key!", k_ast)
```

6.7 Incorrect Comments

Informational **Version 1**

CS-VYPER_APRIL_2024-027

- In `Expr.parse_Hex()` the following comment is incorrect given that bytes_m are right padded with zeros.

```
# bytes_m types are left padded with zeros
```

- In `ir_node_for_log()`, the comment states that the function takes as arguments `buf` and `_maxlen` but it does not.
- In `Stmt._assert_reason()`, the following comment is incorrect given that `method_id` is not abi encoded to `buf+32`:

```
# abi encode method_id + bytestring to `buf+32`, then
# write method_id to `buf` and get out of here
```

- In `Stmt._assert_reason()`, the comment `# offset of bytes in (bytes,)` does not make sense given the commented line:

```
method_id = util.method_id_int("Error(string)")
```

6.8 Missing Case in `clamp_basetype`

Informational **Version 1**

CS-VYPER_APRIL_2024-028

In the case the value to be clamped is an instance of a Flag type having 256 members, `clamp_basetype()` will make the compiler panic as the case is not handled. Note that in practice, the function is currently only called with Flag types needing clamping and this case should not be reachable.

6.9 Optimizations Missing for Transient Storage

Informational **Version 1**

CS-VYPER_APRIL_2024-029

Several optimizations performed when copying data from one location to another have a special path when the source or destination's location is the storage. No equivalent optimization is done for transient storage:

- `make_byte_array_copier()`:
  `has_storage = STORAGE in (src.location, dst.location)`

- `_complex_make_setter()`:
  `has_storage = STORAGE in (left.location, right.location)`

6.10 Unclear Buffer Length Computation

Informational **Version 1**

CS-VYPER_APRIL_2024-030

In `Stmt._assert_reason()`, a buffer of length `64 + msg_ir.typ.memory_bytes_required` is allocated given that the type of the message is `TupleT(StringT[N],)`, this accounts for:

- 32 bytes for the method id.
- 32 bytes for the head of the string according to the ABI.
- `msg_ir.typ.memory_bytes_required` for the String itself.

A similar pattern as what is used in `_pack_arguments()` could be beneficial in showing the different components of the buffer's length.

```
if fn_type.return_type is not None:
    ...
else:
    buflen = args_abi_t.size_bound()

buflen += 32  # padding for the method id
```

6.11 Unreachable Code

Informational **Version 1**

CS-VYPER_APRIL_2024-031

- In `abi_encode()`, the following branch is unreachable given that if `ir_node.typ.is_prim_word`, `abi_encoding_matches_vyper(ir_node.typ)` will always evaluate to true and the "fast path" will be taken before reaching this branch.

```
if ir_node.typ._is_prim_word:
    ir_ret.append(make_setter(dst, ir_node))
```

- In `Expr.handle_external_call()`, the following branch is unreachable given that the logic in `ExprVisitor.visit_Call()` prevents to have an `extcall` or `staticcall` calling a builtin:

```
if isinstance(func_t, BuiltinFunctionT):
    return func_t.build_IR(call_node, context)
```

6.12 Unused Parameters and Variables

Informational **Version 1**

CS-VYPER_APRIL_2024-032

The following parameters and variables are not used:

- clamp in _getelemptr_abi_helper().
- context in handle_binop().
- ret_ofst in _unpack_returndata() (the function is only called by _external_call_helper() which does not use the returned ret_ofst).

6.13 _create_addl_gas_estimate() Omits EIP-3860 Gas Costs

Informational **Version 1**

CS-VYPER_APRIL_2024-033

The function _create_addl_gas_estimate is used to compute the gas used by CREATE and CREATE2 for a given size of the initialization bytecode. The function does however not take into account the change in the gas costs introduced by EIP-3860.