PUBLIC

Limited Code Review

of the Vyper Compiler Back-End Type System

March 31, 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 Assessment Overview section. Pull request #3182 implements a large scale refactoring. It is followed by a general review of the code generation phase of the compiler.

We find that merging the front-end and back-end type systems benefits the code in terms of readability and consistence. Some aspects of the integration of the front-end type system in the code generation are improvable, and introduced bugs, such as StringT not handled in HashMap access.

Other issues have been identified with memory safety, as highlighted by Out of bound memory accesses with DynArray and skip_contract_check skips return data existence check. Special attention should be applied to testing rarely executed codepaths, such as the use of keyword arguments for internal functions, which revealed a long standing bug: Default arguments evaluated incorrectly for internal calls

We recommend being careful with the order of evaluation of expressions. As shown in the case of `DynArrays`, an incorrect evaluation order can lead to bypassing vital safety checks. Regular code reviews can help mitigate the introduction of such issues in the codebase.

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	3
Medium-Severity Findings	2
Low-Severity Findings	8

2 Assessment 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

A review of pull request 3182 of the `https://github.com/vyperlang/vyper` repository was performed. The code differences between the base state of the repository and the resulting state after the pull request has been applied have been examined.

A general best effort review was also performed on the `vyper.codegen` module of the Vyper compiler. The content of the following files was reviewed:

1. vyper/codegen/arithmetic.py
2. vyper/codegen/function_definitions/__init__.py
3. vyper/codegen/function_definitions/external_function.py
4. vyper/codegen/function_definitions/common.py
5. vyper/codegen/function_definitions/utils.py
6. vyper/codegen/function_definitions/internal_function.py
7. vyper/codegen/external_call.py
8. vyper/codegen/self_call.py
9. vyper/codegen/events.py
10. vyper/codegen/ir_node.py
11. vyper/codegen/stmt.py
12. vyper/codegen/__init__.py
13. vyper/codegen/core.py
14. vyper/codegen/memory_allocator.py
15. vyper/codegen/context.py
16. vyper/codegen/return.py
17. vyper/codegen/global_context.py
18. vyper/codegen/module.py
19. vyper/codegen/keccak256_helper.py
20. vyper/codegen/abi_encoder.py
21. vyper/codegen/expr.py

This review was not conducted as an exhaustive search for bugs, but rather as a best effort sanity check for the pull requests of interest.

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. Constant expressions are evaluated.
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. The IR is turned into EVM assembly.
9. Assembly is turned into bytecode.

The review we conducted revolved around phase 7 of the above list, that is code generation phase from the abstract syntax tree.

2.2.1 *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 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 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 key word arguments will generate several selectors, setting the default values for keywords arguments, then call 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:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- 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	3

- Default Arguments Evaluated Incorrectly for Internal Calls
- Out of Bound Memory Accesses With DynArray
- StringT Not Handled in HashMap Access

Medium-Severity Findings	2

- Skip_Contract_Check Skips Return Data Existence Check
- String to Bool Conversion Incorrect

Low-Severity Findings	8

- BytesT to BytesM_T Conversion Can Perform Invalid Memory Access
- Contract With Only Internal Functions Is Executable
- Cost of Memory Expansion for Callees Always Payed
- Dead Code
- IR Labels for Different Functions Can Collide
- Internal Function Arguments Location Set to CALLDATA in Typechecking
- Internal Functions Only Called by __init__ Are Also in the Runtime Code
- parse_type() Can Be Avoided in Favor of Annotations

5.1 Default Arguments Evaluated Incorrectly for Internal Calls

Correctness High Version 1

CS-VYPER_MARCH_2023-001

Internal calls with default arguments are compiled incorrectly. Depending on the number of arguments provided in the call, the defaults are added not right-to-left, but left-to-right. If the types are incompatible, typechecking is bypassed. In the `bar()` function in the following code, `self.foo(13)` is compiled to `self.foo(13,12)` instead of `self.foo(13,1337)`.

```
@internal
def foo(a:uint256 = 12, b:uint256 = 1337):
    pass

@internal
def bar():
    self.foo(13)
```

5.2 Out of Bound Memory Accesses With DynArray

Security **High** **Version 1**

CS-VYPER_MARCH_2023-002

Access to invalid memory can be performed through the use of `DynArray` assignments and mutating operations.

In the following code snippets, uninitialized memory can be read. Since memory frames are reused between function calls, that memory can contain information belonging to other functions.

Literal assignment to ``DynArray``:

```
@internal
def foo():
    c:DynArray[uint256, 1] = []
    c = [c[0]]
```

In the previous example, in line `c = [c[0]]`, the out-of-bound access check is perform after having set the length of c to 1. The check succeeds but the memory is uninitialized.

append:

```
@internal
def foo():
    c:DynArray[uint256, 1] = []
    c.append(c[0])
```

In the previous example, `c.append(c[0])` reads uninitialied memory, but the bounds check succeed because `.append()` increases the length of c before evaluating its arguments.

Furthermore, writing to locations beyond an array length is possible with the use of `pop()`.

pop:

```
@internal
def foo():
    c:DynArray[uint256, 1] = [1]
    c[0] = c.pop()
```

Here the check to write to `c[0]` is performed before the length of the array is reduced by `pop()` to 0. It should revert but it succeeds.

5.3 StringT Not Handled in HashMap Access

Correctness **High** **Version 1**

CS-VYPER_MARCH_2023-003

The code generation for index HashMap index should treat in the same way `StringT` and `BytesT`. The condition at line 337 of `vyper.codegen.expr` only checks `isinstance(index.typ, BytesT)`, instead of `isinstance(index.typ, _BytesArray)`. `BytesLike` got incorrectly turned into `BytesT` in the context of PR3182. As a consequence, the pointer to a string is used to access a HashMap, instead of its hash.

5.4 Skip_Contract_Check Skips Return Data Existence Check

Design Medium Version 1

CS-VYPER_MARCH_2023-004

When calling an external function, the contract existence check can be skipped with the keyword argument `skip_contract_check`. The `skip_contract_check` however also bypasses the checks that the external function call returned the right amount of data, by foregoing the following assert (line 111 of `vyper.codegen.external_call`) :

`["assert", ["ge", "returndatasize", min_return_size]]`

Since the arguments buffer is reused as the return data buffer for the external call, if the called contract does not return data, the unchanged input data is mistaken as the output data of the called function.

As an example, we are calling address `0` with function selector for `f(uint256,uint256)` and arguments 1337 and 6969. The call should revert, because it resulted in no return data, or at most it should return `(0,0)`. However the call returns (1337, 6969). The only reason for this is that the argument buffer is reused as the return buffer.

```
interface A:
    def f(a:uint256, b:uint256) -> (uint256, uint256): view

@external
@view
def foo() -> (uint256, uint256):
    return empty(A).f(1337, 6969, skip_contract_check=True)
```

5.5 String to Bool Conversion Incorrect

Correctness Medium Version 1

CS-VYPER_MARCH_2023-005

The `to_bool` conversion of `_convert.py` accepts `StringT`, which should be treated likely like `BytesT`. However it receives the same treatement as value types, so the pointer is converted to a bool (is zero comparison).

5.6 BytesT to BytesM_T Conversion Can Perform Invalid Memory Access

Design Low Version 1

CS-VYPER_MARCH_2023-006

An out-of-bound memory access is performed, with no consequences, when converting from an empty byte sequence (`b""`) to `bytes32`.

The following code:

```
@internal
def f() -> bytes32:
   return convert(b"", bytes32)
```

generates the following IR:

```
/* convert(b"", bytes32) */
[with,
  arg,
  /* b"" */ [seq, [mstore, 64, 0], 64],
  [with,
    bits,
    [shl, 3, [sub, 32, [mload, arg]]],
    [shl,
      bits,
      [shr, bits, [mload, [add, arg, 32]]]]]]]
```

An `mload` to `arg + 32` is performed, which is out of bounds with respect to the memory size allocated, which is of 1 word for a bytestring of length 0. However, the loaded value is accessed only after shifting it by 256 bits, which means it is zeroed, and its value does not leak to the user.

5.7 Contract With Only Internal Functions Is Executable

Design Low Version 1

CS-VYPER_MARCH_2023-007

If a contract only has internal functions, beside the constructor, it might still compile to executable code. Internals are not pruned, and execution of the internal functions section is not guarded. Upon calling the contract, execution will start at the first internal function. The execution will however generally fail when *POPping* the `RETURN_PC` from the stack, which should be empty upon function exit.

5.8 Cost of Memory Expansion for Callees Always Payed

Design Low Version 1

CS-VYPER_MARCH_2023-008

For functions that perform internal calls conditionally, the gas cost of the memory expansion caused by the internal call is payed even when the internal functions are not called, because the caller memory frame is placed at higher memory addresses than the the callees memory frame.

5.9 Dead Code

Design Low Version 1

CS-VYPER_MARCH_2023-009

Argument `constant_override` of method `FunctionSignature.from_definition` defined in `vyper.ast.signatures.function_signature` is unused throughout the codebase.

Function `parse_Name` in `vyper.codegen.stmt` is likely never executed, as a `Name` can't be a statement. The vdb directive seems to be a left-over from long ago.

5.10 IR Labels for Different Functions Can Collide

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

CS-VYPER_MARCH_2023-010

Labels for `goto` statements are generated in `vyper.ast.signatures.function_signature`. The function `_ir_identifier` is in charge of generating unique IR labels for every function. Depending on the function and the type names, different functions can generate the same labels. IR generation will succeed but assembly generation wil fail.

Example:

```
struct A:
    a:uint256
struct _A:
    a:uint256
@external
@view
def f(b:_A) -> uint256:
   return 1
@external
def f_(b:A):
    pass
```

5.11 Internal Function Arguments Location Set to CALLDATA in Typechecking

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

CS-VYPER_MARCH_2023-011

In `vyper.semantics.analysis.local`, when a `FucntionNodeVisitor` is created, its arguments `DataLocation` are set to `CALLDATA`, regardless if they belong to an internal or external function. For internal functions, the `DataLocation` should be memory. `DataLocation` is not used during code generation, so there are currently no consequences beside a wrong error message when trying to assign an internal function argument.

As an example, compiling the following:

```
@internal
def foo(a:uint256):
    a = 1
```

raises the following exception: `ImmutableViolation: Cannot write to calldata`. In this case we would not be writing to calldata.

5.12 Internal Functions Only Called by `__init__` Are Also in the Runtime Code

Design Low Version 1

CS-VYPER_MARCH_2023-012

Internal functions that are only called in the constructor are still included in the runtime code, increasing its size.

Furthermore, the constructor can call internal functions, but these can't call other internal functions. This is not checked at compile time, but it causes an excecution failure upon deployment.

Example of failing deployment:

```
@external
def __init__():
    self.f()
@internal
def f():
    self.g()

@internal
def g():
    pass
```

5.13 `parse_type()` Can Be Avoided in Favor of Annotations

Design Low Version 1

CS-VYPER_MARCH_2023-013

In the code generation phase, types are parsed from AST objects when they could be recovered from the annotation metadatas added during typechecking.

In `vyper.ast.signatures.function_signature`, at line 135 and 169, the arguments and return type are already contained in `func_ast._metadata['type']`, which is an instance of `ContractFunctionT`.

In *parse_AnnAssign*` in `vyper.codegen.stmt`, the type could be stored in the AST node during local function analysis (`AnnAssign` nodes currently do not store `_metadata['type']`).

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 Default Return Value Evaluated Conditionally

Note Version 1

Calling an external function with `default_return_value=a()` will only evaluate `a()` after the external call has been performed, if the call resulted in no return data. This behavior is undocumented and clashes with the usual semantics, where all arguments are evaluated.

6.2 Order of Evaluation of Event Arguments

Note Version 1

As in solidity, the order of evaluation of event arguments in Vyper is counter-intuitive, and doesn't follow the usual conventions. First the indexed parameters are evaluated right to left, then the non-indexed parameters are evaluated left to right.

In the example, the internal calls are performed in the order self.a(), self.b(), self.c(), self.d():

```
event A:
    b:indexed(uint256)
    c:uint256
    d:uint256
    a:indexed(uint256)

@internal
def a() -> uint256:
    return 1
@internal
def b() -> uint256:
    return 2
@internal
def c() -> uint256:
    return 3
@internal
def d() -> uint256:
    return 4

@internal
def foo():
    log A(self.b(), self.c(), self.d(), self.a())
```

This unusual behavior should be highlighted.