PUBLIC

Limited Code Review

of the Vyper Compiler 0.4.3 Pull requests

June 16, 2025

Produced for

by

Contents

1 Executive Summary

Dear Vyper team,

Thank you for trusting us to help Vyper with this security 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 do not provide assurance comparable to non-limited code assessments or reviews. Due to time and scope constraints, they are not exhaustive. This limited review was conducted by one engineer over two days and focused on the several pull requests fixing known issues, and new `raw_return` functionality that will ease proxy implementations in Vyper.

The most critical subjects addressed in our review include the correctness of both the implementation of the new `raw_return` functionality and the fixes applied, particularly those related to side effects evaluation and the `raw_create` built-in function. Our assessment found that all fixes were implemented correctly and that `raw_return` operates as intended.

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	0
Low-Severity Findings	1

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

This review was not conducted as an exhaustive search, but instead focused on fixes implemented for issues already tracked on GitHub, specifically the following pull requests:

- https://github.com/vyperlang/vyper/pull/4645
- https://github.com/vyperlang/vyper/pull/4649
- https://github.com/vyperlang/vyper/pull/4644
- https://github.com/vyperlang/vyper/pull/4621
- https://github.com/vyperlang/vyper/pull/4619
- https://github.com/vyperlang/vyper/pull/4623
- https://github.com/vyperlang/vyper/pull/4622
- https://github.com/vyperlang/vyper/pull/4624
- https://github.com/vyperlang/vyper/pull/4632

As well as the new `raw_return` functionality:

- https://github.com/vyperlang/vyper/pull/4568

For pull request already merged into master, the review looked at the changes introduced by them and their impact at the commit described in the table below, for not-yet-merged pull requests, their diff with the merge target was reviewed. Issues already tracked on GitHub were not included in this report.

The assessment was performed on the source code files inside the Vyper Compiler repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	07 June 2025	f2c4d1384412073568695f2d5797009c4c79549d	Initial Version

2.1.1 Excluded from scope

2.2 System Overview

The Vyper language is a pythonic smart-contract-oriented language targeting the Ethereum Virtual Machine (EVM). Designed with security as a primary goal, Vyper deliberately omits features found in other languages like recursion, infinite loops, and inheritance.

The Vyper compiler transforms source code into EVM bytecode through a well-defined compilation pipeline:

1. **Parsing**: Vyper source code is parsed into an Abstract Syntax Tree (AST) that represents the syntactic structure of the program.

2. **Literal Validation**: Literal nodes in the AST (constants, strings, numbers, etc.) are validated to ensure they conform to expected formats and ranges.
3. **Semantic Analysis**: The program's semantics are validated while populating the namespace. The structure and types are checked, type annotations are added to the AST, module imports are resolved, and related properties are verified.
4. **Layouts**: Storage slots are assigned to storage variables and data locations are allocated for immutable variables.
5. **IR Generation**: The typed and validated AST is transformed into a lower-level intermediate representation (IR) that more closely resembles EVM operations.
6. **Optimization**: Various optimization passes are applied to the IR to improve gas efficiency and/or reduce code size.
7. **Assembly Generation**: The IR is converted to EVM assembly instructions.
8. **Bytecode Generation**: Assembly is translated into bytecode, 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 undergoes comprehensive analysis to verify compliance with Vyper semantics. This process annotates the AST with types and metadata necessary for the code generation phase.

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 Open Findings

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

- Design: Architectural shortcomings and design inefficiencies

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	0
Low-Severity Findings	1

5.1 `IfExp` Nodes Incorrectly Parsed to IR Node

Design Low Version 1

CS-VYPER_JUNE_2025-001

`IfExp` AST nodes containing on both side values that have no location compile to an incorrect IR node.

This is due to the following branch in `parse_IfExp` not being entered as both side of the `IfExp` have no location.

```
if body.location != orelse.location or body.value == "multi":
    body = ensure_in_memory(body, self.context)
    orelse = ensure_in_memory(orelse, self.context)
```

The branch can be bypassed if either:

- The two side are "empty" values, i.e. `empty(T)` (see static array example below).
- The left side is an "empty" value, and the right side is a "literal" (multi) value. (see dynamic array example below).

Note that this issue extends issue #3480 tracked on GitHub.

Static arrays:

```
@external
def test():
    a : uint256[256] = empty(uint256[256]) if True else empty(uint256[256])
```

```
vyper.exceptions.CompilerPanic: unreachable!
```

Dynamic arrays:

```
@external
def test():
    a: DynArray[uint256, 256] = [] if True else []
    b: DynArray[uint256, 256] = empty(DynArray[uint256, 256]) if True else [1]
```

```
vyper.exceptions.CompilerPanic: cannot dereference non-pointer type
```

Structs:

```
struct S:
    a: bool

@external
def test():
    a: S = empty(S) if True else empty(S)
```

```
vyper.exceptions.CompilerPanic: unreachable!
```

Tuples:

```
@external
def test():
    a: uint256 = 0
    b: uint256 = 0
    a, b = empty((uint256, uint256)) if True else empty((uint256, uint256))
```

```
vyper.exceptions.CompilerPanic: unreachable!
```

Bytes and Strings:

```
@external
def test():
    a: Bytes[32] = empty(Bytes[32]) if True else empty(Bytes[32])
```

```
vyper.exceptions.CodegenPanic: unhandled exception 'NoneType' object has no attribute 'has_copy_opcode', parse_AnnAssign
```

uint256:

```
@external
def test():
    a: uint256 = empty(uint256) if True else 1
```

```
ValueError: Weird code element: <class 'vyper.codegen.ir_node.IRnode'> ~empty <empty(uint256)>
```

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 Constant and Immutable Can Be Annotated With `reentrant`

Informational **Version 1**

CS-VYPER_JUNE_2025-002

PR #4622 make public getter for `constant` and `immutable` variables always reentrant as they cannot be used for a read only reentrancy attack, due to their nature of being constant at runtime. It is however still possible to annotate them with `reentrant`, which have no effect on the code generation:

```
# pragma nonreentrancy on
a: public(reentrant(constant(uint256))) = 1
```

6.2 Slice Out of Bounds Check Off by One

Informational **Version 1**

CS-VYPER_JUNE_2025-003

In `Slice.fetch_call_return`, the following bounds check contains an off-by-one error that makes it less precise than it could be:

```
if start_literal is not None:
    if start_literal > arg_type.length:
        raise ArgumentException(f"slice out of bounds for {arg_type}", start_expr)
    if length_literal is not None and start_literal + length_literal > arg_type.length:
        raise ArgumentException(f"slice out of bounds for {arg_type}", node)
```

The condition `start_literal > arg_type.length` could be made more precise by checking `start_literal >= arg_type.length` instead. This is because a slice is out of bounds when the start index is equal to the length of the array, not just when it exceeds the length.

6.3 Stack Management Inconsistency

Informational **Version 1**

CS-VYPER_JUNE_2025-004

There is a fundamental inconsistency in how stack cleanup is handled between different control flow mechanisms in Vyper's IR compilation. The `break` statement assumes that stack items can accumulate after evaluation of other IR statements, while `cleanup_repeat` and `exit_to` assume no such accumulation occurs, and only `repeat` can leave items on the stack.

The `break` statement explicitly handles the case where `height - break_height > 0`, implicitly meaning that it expects there to be items on the stack that need to be cleaned up:

```
elif code.value == "break":
    if not break_dest:
        raise CompilerPanic("Invalid break")
    dest, continue_dest, break_height = break_dest

    n_local_vars = height - break_height
    # clean up any stack items declared in the loop body
    cleanup_local_vars = ["POP"] * n_local_vars
    return cleanup_local_vars + [dest, "JUMP"]
```

In contrast, the `cleanup_repeat` and `exit_to` instructions do not account for any stack items that may have been left on the stack by previous operations. They only handle the case where the stack contains items from the `repeat` loop itself:

```
elif code.value == "cleanup_repeat":
    if not break_dest:
        raise CompilerPanic("Invalid break")
    # clean up local vars and internal loop vars
    _, _, break_height = break_dest
    # except don't pop label params
    if "return_buffer" in withargs:
        break_height -= 1
    if "return_pc" in withargs:
        break_height -= 1
    return ["POP"] * break_height
```

In the current implementation, this inconsistency is not problematic because statements cannot leave items on the stack after their evaluation. The stack is always cleaned up before the next statement, or a child statement is executed (except for `repeat`), ensuring that the stack state remains consistent. If in the future, stack items are allowed to accumulate after the evaluation of other IR statements, this inconsistency could lead to several issues:

1. Stack Corruption: `cleanup_repeat` or `exit_to` would leave stack items unpopped, corrupting the stack state for outer functions.
2. Runtime Failures: Stack underflow/overflow in complex nested scenarios
3. Inconsistent Behavior: Different control flow paths would handle stack cleanup differently