PUBLIC

Limited Code Review

of the Vyper Compiler 0.4.2 Pull Requests

April 24, 2025

Produced for

by

Contents

1 Executive Summary

Dear Vyper team,

Thank you for trusting us to help Vyper with this limited security review. Our executive summary provides an overview of subjects covered in our review of the latest version of the 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 non-limited code assessments or audits. Due to time and scope constraints, they are not exhaustive. This limited review was conducted by one engineer over one week and focused on multiple pull requests of the to-be-released version 0.4.2 of the Vyper compiler.

The most critical subjects covered in our review are the non-reentrancy by default option and the `raw_create` builtin. Security regarding all the aforementioned subjects is high. Moreover, we found that allowing users to turn on the non-reentrancy by default option is a good security measure that benefits language users greatly.

Other general subjects covered include enabling bitwise operators for `bytesM`, extending `as_wei_value` to all numeric types, refactoring decorator and pragma parsing as well as other smaller pull requests. Security regarding all the aforementioned subjects is also high.

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	10

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

The review consisted of a 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/4563
- https://github.com/vyperlang/vyper/pull/4204
- https://github.com/vyperlang/vyper/pull/4538
- https://github.com/vyperlang/vyper/pull/4419
- https://github.com/vyperlang/vyper/pull/3498
- https://github.com/vyperlang/vyper/pull/4540
- https://github.com/vyperlang/vyper/pull/4574
- https://github.com/vyperlang/vyper/pull/4156
- https://github.com/vyperlang/vyper/pull/4490
- https://github.com/vyperlang/vyper/pull/4536
- https://github.com/vyperlang/vyper/pull/4571
- https://github.com/vyperlang/vyper/pull/3157
- https://github.com/vyperlang/vyper/pull/4530
- https://github.com/vyperlang/vyper/pull/4566
- https://github.com/vyperlang/vyper/pull/4550
- https://github.com/vyperlang/vyper/pull/4371
- https://github.com/vyperlang/vyper/pull/4492

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 master was reviewed. Issues already tracked on GitHub or fixed in https://github.com/charles-cooper/vyper/pull/79/commits/8329221897cd4f9239ac736f6d20624bc4ba37c6 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	15 April 2025	79001f3e0aae98058ff7c43c3c930d2100751ebd	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
- 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	0
Low-Severity Findings	10

5.1 Builtin Mutability Not Checked at Semantic Analysis

Design Low Version 1

CS-VYPER_APRIL_2025-001

Pull request #3157 disallow `blockhash` and `blobhash` builtins in pure functions. Similarly, when trying to call state modifying builtins from a pure or view function, the compiler fails with a `StateAccessViolation` error:

```
@external
@view
def foo(x: uint256):
    z: address = raw_create(b'')
```

```
vyper.exceptions.StateAccessViolation: Cannot use raw_create from a constant function
```

However, as opposed to `blockhash` and `blobhash`, the error here is triggered during the code generation of the builtin's `build_IR`, and not earlier at semantic analysis:

```
@process_inputs
def build_IR(self, expr, args, kwargs, context):
    # errmsg something like f"Cannot use {self._id} in pure fn"
    context.check_is_not_constant(f"use {self._id}", expr)
```

Very similarly to what was done for `blockhash` and `blobhash`, the error could be raised earlier at semantic analysis.

5.2 Builtins Variable Arguments Are Not Semantically Checked

Design Low Version 1

CS-VYPER_APRIL_2025-002

Builtins that accept variable arguments (varargs) do not semantically checked them. In case incorrect object are passed (types, environment reserved keywwords, etc.), the compiler will fail for diverse reason only at code generation.

The following code shows different examples of this issue (`lib1` is an empty module here):

```
import lib1

initializes: lib1

a: HashMap[uint256, address]

@external
def foo():
    # CodegenPanic: unhandled exception 'IntegerT' object has
    #               no attribute 'location', parse_Call
    a: address = raw_create(b'hello world', uint256)
    # TypeCheckFailure: assigning HashMap[uint256, address] to
    #                   HashMap[uint256, address] 160 <#internal1> 0 <self.a>
    b: address = raw_create(b'hello world', self.a)
    # CompilerPanic: Method must be implemented by the inherited class
    c: Bytes[1000] = abi_encode(msg)
    # CompilerPanic: Method must be implemented by the inherited class
    d: Bytes[1000] = abi_encode(lib1)
```

5.3 Codegen Panic When Converting Literal Bytes to Decimal

Correctness Low Version 1

CS-VYPER_APRIL_2025-003

In `_convert.py`, `to_decimal()` is annotated with `@_input_types(IntegerT, BoolT, BytesM_T, BytesT)`, which means that it can accept `Bytes` and `HexBytes` nodes. However, when calling `_literal_decimal()`, the compiler will crash as the function does not handle any of these nodes:

```
@_input_types(IntegerT, BoolT, BytesM_T, BytesT)
def to_decimal(expr, arg, out_typ):
    _check_bytes(expr, arg, out_typ, 32)

    if isinstance(expr, vy_ast.Constant):
        return _literal_decimal(expr, arg.typ, out_typ)

    ...
```

This happens as trying to construct a `Decimal` from a `bytes` is not supported:

```
val = decimal.Decimal(expr.value)  # should work for Int, Decimal
```

The following code lead the compiler to panic:

```
# pragma enable-decimals
@external
def foo():
    a:decimal = convert(b'\x01\x02', decimal)
    b:decimal = convert(x'0102', decimal)
```

5.4 Inconsistent Lark Grammar for Constants

Design | Low | Version 1

CS-VYPER_APRIL_2025-004

In `grammar.lark`, the grammar for constants is defined as:

```
constant: "constant" "(" type ")"
constant_private: ChainSecurity ":" constant
constant_with_getter: ChainSecurity ":" "public" "(" constant ")"
constant_def: (constant_private | constant_with_getter) "=" expr
```

However, this does not account for the `reentrant` decorator, which is a valid decorator for constant variables, since the below code is valid Vyper code, but fails to be parsed using the lark grammar:

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

5.5 Layout Export Omits Non-Reentrant Key Slot

Design | Low | Version 1

CS-VYPER_APRIL_2025-005

In `data_position.py`, the function `_generate_layout_export_r()` generates the storage, transient and code layout for the contract. For the reentrancy key location, the following is done:

```
for fn in vyper_module.get_children(vy_ast.FunctionDef):
    fn_t = fn._metadata["func_type"]
    if not fn_t.nonreentrant:
        continue
```

```
    location = get_reentrancy_key_location()
    layout_key = _LAYOUT_KEYS[location]

    if GLOBAL_NONREENTRANT_KEY in ret[layout_key]:
        break

    slot = fn_t.reentrancy_key_position.position
    ret[layout_key][GLOBAL_NONREENTRANT_KEY] = {
        "type": "nonreentrant lock",
        "slot": slot,
        "n_slots": NONREENTRANT_KEY_SIZE,
    }
    break
```

However, in the case no nonreentrant functions are present in the contract, but only public state variables with the `pragma nonreentrancy on`, the `_generate_layout_export_r()` function will not output the slot for the key, although it is allocated with `_allocate_layout_r()`.

The following contracts yield the given layout when running `vyper -f layout file.vy`, although the transient storage slot 0 is allocated and used by the public getter function for the a variable:

```
# pragma nonreentrancy on

a: public(constant(uint256)) = 1

b: transient(uint256)
```

```
{"transient_storage_layout": {"b": {"type": "uint256", "n_slots": 1, "slot": 1}}}
```

5.6 Memory Buffer Invalidated Before Read

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

CS-VYPER_APRIL_2025-006

The `raw_create` builtin takes as input a byte array for the initcode, constructor arguments, and both the value and salt. Since each argument's evaluation might have side effect that can change other arguments, they are ensured to be in memory (`ensure_in_memory()`) and/or cached (`scope_multi()`, `cache_when_complex()`) prior to logic execution:

```
def _build_create_IR(self, expr, args, context, value, salt, revert_on_failure):
    args = [ensure_in_memory(arg, context) for arg in args]
    initcode = args[0]
    ctor_args = args[1:]

    # encode the varargs
    to_encode = ir_tuple_from_args(ctor_args)
    type_size_bound = to_encode.typ.abi_type.size_bound()
    bufsz = initcode.typ.maxlen + type_size_bound

    buf = context.new_internal_variable(get_type_for_exact_size(bufsz))

    ret = ["seq"]
```

```
    with scope_multi((initcode, value, salt), ("initcode", "value", "salt")) as (
        b1,
        (initcode, value, salt),
    ):
        bytecode_len = get_bytearray_length(initcode)
        with bytecode_len.cache_when_complex("initcode_len") as (b2, bytecode_len):
            maxlen = initcode.typ.maxlen
            ret.append(copy_bytes(buf, bytes_data_ptr(initcode), bytecode_len, maxlen))

            argbuf = add_ofst(buf, bytecode_len)
            argslen = abi_encode(
                argbuf, to_encode, context, bufsz=type_size_bound, returns_len=True
            )
            total_len = add_ofst(bytecode_len, argslen)
            ret.append(_create_ir(value, buf, total_len, salt, revert_on_failure))

            return b1.resolve(b2.resolve(IRnode.from_list(ret)))
```

These protections miss one edge case when the bytecode is stored in memory, and the evaluation of another argument invalidates the memory location of the bytecode. This is the case for example when the bytecode is stored in a dynamic array, and another argument pop from that array. The following code shows this issue:

```
@external
def deploy_from_calldata(s: Bytes[49152]) -> address:
    v: DynArray[Bytes[49152], 2] = [s]
    x: address = raw_create(v[0], value = 0 if v.pop() == b'' else 0)
    return x
```

Behind the scene, the following actions are done:

1. The pointer to `v[0]` is cached (`scope_multi`: `initcode`)
2. `v.pop()` is evaluated, which pop the first element of the array and invalidate the pointer to `v[0]`. (`scope_multi`: `value`)
3. The cached pointer is used to read the bytecode, but it is now invalid. (`bytecode_len()`, `bytes_data_ptr()`)

Given the implementation of `pop()`, the value of `v[0]` is not changed, and the step 3 will read the correct value still, but this is not guaranteed in the future as other expression that modify in place memory variables might get implemented.

This issue is not be limited to `raw_create()`, but more generally applies to anywhere the compiler relies on `ensure_in_memory()` to cache a pointer to a non-primitive type in order to prevent other expressions to modify it before it is read.

5.7 Missing Execution Paths for `HexBytes`

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

CS-VYPER_APRIL_2025-007

The AST node `HexBytes` is used to represent byte arrays in the Vyper AST, similar to the `Bytes` node, but with a hexadecimal representation. There should be no reason to have different behavior for these two nodes, but the `HexBytes` node is not always handled in the same way as the `Bytes` node.

1. In `_convert.py::_literal_int()`, there are no paths to handle the `HexBytes` node, which means that upon converting a `HexBytes` node to an integer (`convert(x'0102', uint256)`),

the compiler will fail with `CompilerPanic: unreachable`. Furthermore, in case the out type is signed, the HexBytes value would not sign-extended if the first issue was fixed.

```
def _literal_int(expr, arg_typ, out_typ):
    # TODO: possible to reuse machinery from expr.py?
    if isinstance(expr, vy_ast.Hex):
        val = int(expr.value, 16)
    elif isinstance(expr, vy_ast.Bytes): # Missing HexBytes
        val = int.from_bytes(expr.value, "big")
    elif isinstance(expr, (vy_ast.Int, vy_ast.Decimal, vy_ast.NameConstant)):
        val = expr.value
    else:  # pragma: no cover
        raise CompilerPanic("unreachable")

    # Missing HexBytes
    if isinstance(expr, (vy_ast.Hex, vy_ast.Bytes)) and out_typ.is_signed:
        val = _signextend(expr, val, arg_typ)
```

2. Similarly, in `_convert.py::_literal_decimal()`, in case the out type is signed, the HexBytes value is not sign-extended, note that in the current state of the codebase, this is not reachable because of Codegen panic when Converting literal bytes to decimal:

```
if isinstance(expr, (vy_ast.Hex, vy_ast.Bytes)) and out_typ.is_signed:
    val = _signextend(expr, val, arg_typ)
```

3. In the `Len` builtin defined in `functions.py`, a special path to constant fold `Bytes` nodes is defined, but not for `HexBytes` nodes. This means that the following code will not be folded:

```
@external
def foo():
    a:uint256 = len(x'0102')
```

5.8 Pragmas Can Be Defined Anywhere

Design | Low | Version 1

CS-VYPER_APRIL_2025-008

Using the notation `# pragma <name> [<value>]`, pragmas can be specified in contracts to indicate specific behavior requested from the compiler, such as compiler version, optimizations, or EVM version.

The `PreParser` class is responsible for parsing the pragma statement, and it does so by looking for comments in the source code that start with `pragma` (or `@version` for the outdated way of specifying compiler version).

However, there is no check to ensure that the pragma statement is at the beginning of the file, outside any function, or type definition. This means that the following code is considered valid by the compiler and the pragma statements are considered:

```
struct A:
    # pragma version 0.4.2
    a: uint256

@external
def foo():
    # pragma nonreentrancy on
    pass
```

5.9 State Variable Annotation Can Appear Twice

Correctness Low Version 1

CS-VYPER_APRIL_2025-009

On top of `immutable`, `constant` and `transient`, state variables can be annotated with `reentrant` or `public` or both.

The `__init__` function of the `VariableDecl` class implements the logic to check for the presence of these annotations as follows:

```
for _ in range(2):
    func_id = self.annotation.get("func.id")
    if func_id not in ("public", "reentrant"):
        break
    _check_args(self.annotation, func_id)
    setattr(self, f"is_{func_id}", True)
    # unwrap one layer
    self.annotation = self.annotation.args[0]
```

Since there is no check to ensure that an annotation can only appear once, the following code is considered valid by the compiler:

```
a: public(public(uint256))
```

5.10 State Variable Annotation Can Have Kwargs

Correctness Low Version 1

CS-VYPER_APRIL_2025-010

State variable declaration can be annotated with combinations of `immutable`, `constant`, `transient`, `reentrant` or `public`.

In practice, they are parsed as AST `Call` nodes:

```
# the annotation is a "function" call, e.g.
# `foo: public(constant(uint256))`
# pretend we were parsing actual Vyper AST. annotation would be
# TYPE | PUBLIC "(" TYPE | ((IMMUTABLE | CONSTANT) "(" TYPE ")") ")"
```

This means that restrictions on the arguments of the annotation are not enforced by the parser, but rather by the compiler itself, this is done in the `VariableDecl` class, which is responsible for checking the arguments of the annotation:

```
def _check_args(annotation, call_name):
# do the same thing as `validate_call_args`
# (can't be imported due to cyclic dependency)
if len(annotation.args) != 1:
    raise ArgumentException(f"Invalid number of arguments to `{call_name}`:", self)
```

However, the parser does not check for the presence of keyword arguments in the annotation, which means that the following code is considered valid by the compiler:

```
a: public(uint256, a = hello, b = 1212)
```

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 Bitwise Operations on `bytesM` Types Are Not Folded

Informational **Version 1**

CS-VYPER_APRIL_2025-011

Bitwise operations can be performed on `bytesM` types. However, unlike bitwise operations on numeric types, folding is not performed for `bytesM` types.

```
def visit_UnaryOp(self, node):
    operand = node.operand.get_folded_value()

    ...

    if isinstance(node.op, vy_ast.Invert) and not isinstance(operand, vy_ast.Int):
        raise UnfoldableNode("not an int!", node.operand)

    ...

def visit_BinOp(self, node):
    ...

    if not isinstance(left, vy_ast.Num):
        raise UnfoldableNode("not a number!", node.left)

    ...
```

For example, in the following code, the bitwise & operator is not folded, and will be evaluated at runtime:

```
# pragma optimize none

a:constant(bytes1) = 0x21 & 0x14
@external
def foo():
    l: bytes1 = a
```

Note that the optimizer should generally be able to optimize such code. Missing folding might be an issue if semantic analysis depends on the folding of the operation, which is however uncommon for `bytesM` types.

6.2 Code Refactoring Opportunities

Informational **Version 1**

CS-VYPER_APRIL_2025-012

- In `function.py::_parse_decorators()`, the for loop iterates over the decorators and pattern match over the AST node type, and, if the node is a `Name`, its `id` attribute. The first two cases for `nonreentrant` and `reentrant` could be handled together with the general case for `Name` nodes to enhance readability as they currently implicitly assume that the decorator is a `Name` node with `.get("id")`.

```
def _parse_decorators(funcdef: vy_ast.FunctionDef) -> _ParsedDecorators:
    ret = _ParsedDecorators(funcdef)

    for decorator in funcdef.decorator_list:
        # order of precedence for error checking
        if decorator.get("id") == "nonreentrant":
            ret.set_nonreentrant(decorator)

        elif decorator.get("id") == "reentrant":
            ret.set_reentrant(decorator)

        elif isinstance(decorator, vy_ast.Call):
            ...
            raise StructureException(msg, decorator, hint=hint)

        elif isinstance(decorator, vy_ast.Name):
            if FunctionVisibility.is_valid_value(decorator.id):
                ret.set_visibility(decorator)
            elif StateMutability.is_valid_value(decorator.id):
                ret.set_state_mutability(decorator)
            else:
                raise FunctionDeclarationException(f"Unknown decorator: {decorator.id}", decorator)
```

- In `function.py::_ParsedDecorators`, the `set_visibility()` function could use the `_check_none()` function to check if the visibility was already set to be consistent with `set_state_mutability()`, `set_nonreentrant()`, and `set_reentrant()`.
- In `core.py::_prefer_copy_maxbound_heuristic()`, the following is computed: `ceil32(src_bound - 32) * 3 // 32` and assumes that `src_bound` is always greater than or equal to 32. While this is true, it would be beneficial to add an assert statement or a comment to clarify this assumption.
- In `test_bitwise.py::test_bitwise_opcodes()`, the following assertion will always suceed since SHR is always used by vyper contract in the function selector.

```
assert "SHR" in opcodes
```

6.3 Constructor Can Be Marked as Reentrant

Informational **Version 1**

CS-VYPER_APRIL_2025-013

In theory, constructors cannot be reentered, as when being executed, the contract still has no bytecode. For this reason, Vyper prevents adding a `@nonreentrant` decorator on `__init__()` functions. However, in case `nonreentrancy` is set to on, it is possible to mark them as reentrant, which will not alter the code generated, but could be misleading to the user. The following code is considered valid by the compiler:

```
# pragma nonreentrancy on

@deploy
@reentrant
def __init__():
    pass
```

6.4 Misleading Comments

Informational **Version 1**

CS-VYPER_APRIL_2025-014

The code contains misleading comments that do not accurately describe the functionality or behavior of the code.

- In `core.py::make_byte_array_copier()`, the following comment is misleading as a byte array is copied and not a dynamic array:

```
# batch copy the entire dynarray, including length word
```

- In `core.py::_prefer_copy_maxbound_heuristic()`, the following comment suggests that 5 to 8 bytes are saved with the heuristic, when in fact it is 3 to 6 bytes since using the bound still need a `PUSH1 BOUND` instruction, which uses 2 bytes:

```
# if we are optimizing for codesize, we are ok with a higher
# gas cost before switching to copy(dst, src, <precise length>).
# +45 is based on vibes -- it says we are willing to burn 45
# gas (additional 15 words in the copy operation) at runtime to
# save these 5-8 bytes (depending on if itemsize is 1 or not)
```

- In `expr.py::handle_binop()`, the following comment could also mention `bytes`, that are handled earlier in the function, similarly to `flags`:

```
# flags can only do bit ops, not arithmetic.
```

- In `expr.py::parse_UnaryOp()`, the following comment is misleading as not only `not` is a supported operator, but also ~ and `-`:

```
# Unary operations (only "not" supported)
```

6.5 Misleading Error Messages

Informational **Version 1**

CS-VYPER_APRIL_2025-015

- When trying to call an internal function marked as `@nonreetrant` from a function itself non-reentrant, the following error message is shown:

```
msg = f"Cannot call `{g.name}` since it is"
msg += f" `@nonreentrant` and reachable from `{fn_t.name}`"
msg += ", which is also marked `@nonreentrant`"
raise CallViolation(msg, func, g.ast_def)
```

  In case the `pragma nonreentrancy on` is set, the error message is misleading as it suggest that the calling function has a non-reentrant decorator, when in fact it does not necessarily as shown in the following code:

```
# pragma nonreentrancy on

@external
```

```
def bar():
    self.foo()

@nonreentrant
def foo():
    pass
```

- When trying to use the `blockhash` or `blobhash` built-in functions in a pure function, the following error message is shown, it could be improved to clarify that the given built-in is not allowed:

```
StateAccessViolation: Cannot call a view function from a pure function
```

6.6 Pure Functions Can Be Marked as Reentrant

Informational **Version 1**

CS-VYPER_APRIL_2025-016

Pure functions cannot write or read from state, which means that they cannot implement any reentrancy protection. For this reason, they cannot be marked as `@nonreentrant`. However, in case `nonreentrancy` is set to `on`, it is possible to mark them as reentrant, which will not alter the code generated, but could be misleading to the user. The following code is considered valid by the compiler:

```
# pragma nonreentrancy on

@reentrant
@pure
def foo():
    pass
```

6.7 Unused Code

Informational **Version 1**

CS-VYPER_APRIL_2025-017

In `core.py`, the function `clamp_nonzero()` is defined but never used:

```
def clamp_nonzero(arg):
    # TODO: use clamp("ne", arg, 0) once optimizer rules can handle it
    with IRnode.from_list(arg).cache_when_complex("should_nonzero") as (b1, arg):
        check = IRnode.from_list(["assert", arg], error_msg="check nonzero")
        ret = ["seq", check, arg]
        return IRnode.from_list(b1.resolve(ret), typ=arg.typ)
```

7 Notes

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

7.1 Internal Functions Can Be Marked as `@reentrant` Even When `nonreentrancy on` Is Set

Note Version 1

When `pragma nonreentrancy on` is set, internal functions defaults to being reentrant, it is still possible to mark them as `@reentrant`, and this will have no effect on the compilation.

The following code is valid Vyper code, and the `@reentrant` decorator will be ignored by the compiler:

```
# pragma nonreentrancy on

@reentrant
def foo():
    pass
```

7.2 Reentrancy Key Location

Note Version 1

A module and its sub(sub)module(s) might have different pragma set for the `evm-version`. It should be noted that in such case, for the global reentrancy key location (`storage` or `transient`), the pragma of the outermost (compilation target) is always used. That is:

- If a module with evm version `cancun` imports a submodule with evm version `shanghai`, the global reentrancy key location will be `transient`.
- If a module with evm version `shanghai` imports a submodule with evm version `cancun`, the global reentrancy key location will be `storage`.