CONSENSYS
Diligence

Vyper Security Review

Date	October 2019
Reviewers	Steve Marx, Todd Proebsting

1 Scope

ConsenSys Diligence conducted a preliminary review of beta 13 of the Vyper compiler. The goal of this review was to identify what work remains before the compiler is ready for a full security audit.

2 Executive Summary

Auditing code requires comparing the code to expected behavior. For a compiler, the expected behavior is determined by the source language specification and the target

machine. To confidently audit a Vyper compiler, a complete and precise Vyper specification is needed.

The Vyper compiler is currently decomposed into three passes: a small pass that translates Python abstract syntax trees to Vyper ASTs, a large pass that translates Vyper ASTs to LLL intermediate representation, and a small pass that translates LLL to EVM bytecode. The AST→LLL pass is too big and complex to be confidently audited. We recommend decomposing it into smaller passes with well-defined interfaces. This decomposition will allow independent auditing of each pass.

During the review, we found a number of bugs, which are described at the end of this document.

3 Compiler Decomposition

Logically, compilation can be viewed as a sequence of transformations that start with source code and end with binary code, with different intermediate representations between each pass. A canonical (non-optimizing) compiler decomposition might be the following:

1. *Lexical Analysis*: Transforms raw characters into “tokens”. E.g., keywords, variables, operators, etc.
2. *Parsing*: Transforms tokens into abstract syntax trees (ASTs). The tree nodes represent the constructs of the language. E.g., for-loops, if-then-elses, function declarations, expressions.
3. *Semantic Analysis*: Determines all the semantic information (a.k.a. “meaning”) of the represented program. This pass will typically do many of the following:
    1. Track the declaration and uses of every identifier in the program, and any necessary information about each identifier. E.g., the type of variables, the scope of variables, etc.
    2. Determine the types of all expressions, including any implicit type conversions. Type checking for expressions includes inferring result types based on operand types, checking that literals conform to Vyper or EVM limitations, checking variable/name usage, etc. (`parser/expr.py` reports errors in 67 different locations.)
    3. Annotate the ASTs with the type information that is computed. For instance, after checking that the types of the left and right operands of an ADD node can be added together, it might annotate the ADD node with the type it will compute.

    4. Make explicit what was implicit. Semantic analysis might modify the AST to make previously implicit operations explicit. For instance, it may add a type conversion to widen an integer value that is an operand to an ADD where the other operand is wider. Similarly, it might tag a literal value with how it is going to be used. For instance, it might tag the literal `7` as a `uint256` if it is the operand of an ADD where the other operand is known to be a `uint256`.
4. *Constant Folding*: Transforms the ASTs based on operations that can be computed at compile time. E.g., `1+2` can be replaced with `3`.
5. *Intermediate Representation (IR) Generation*: Translates all of the fully analyzed (and annotated) ASTs into a chosen IR.
6. *Code Generation*: Translate the IR into the target code (i.e., EVM bytecode).

The benefit of decomposition is that each pass can be analyzed independently. Of course, this requires that the data structures consumed and generated by each pass be well-defined. If we ignore lexical analysis (which is done for Vyper by the Python parser) the decomposition above requires only two well-defined data structures: the AST and the IR. (The data structure that represents types will, of course, annotate the AST.)

4 Abstract Syntax Trees (AST)

The Vyper compiler's AST is **not** an abstract syntax tree for Vyper—it is an AST for Python. The Vyper compiler translates the Python compiler-generated AST nodes into its own AST nodes, but those nodes are almost perfect replicas of the original nodes. Thus, the Vyper compiler is burdened with interpreting Python-like abstract syntax as if it were Vyper syntax.

A good example of this is the AST nodes for Vyper's `struct` and `contract` constructs. There aren't any. Instead, Vyper overloads Python's `class` AST node to represent structs and classes, distinguishing them with a special field.

There are many examples of where Python's AST is used to (unnaturally) represent Vyper syntax:

1. Using Python function call ASTs to represent the following Vyper constructs:
    1. events
    2. type modifiers (`constant` , `public`)
    3. type units (e.g. `uint256(wei)`)
    4. maps

2. Using Python class ASTs to represent:
    1. structs
    2. contracts
3. Using Python's general for-loop AST to represent Vyper's restricted for-loop
4. Using Python's annotated assignment AST to represent
    1. `implements`
    2. type declarations
    3. custom units declarations
5. Using Python's decorator AST for Vyper's function modifiers (`@nonreentrant`, `@public`, etc.)

5 Separate Type Checking Pass

Vyper's static type system is sufficiently complex to merit its own separate pass in the compiler. This would separate the semantic analysis involved in checking and inferring type information from the process of generating LLL. The current integration of those concerns makes understanding either process more difficult and, possibly, more error-prone. (It may also be hiding opportunities for factoring out common type checking routines.)

A type-checking pass in the compiler would typically annotate AST nodes with type information. This is especially useful for expressions and built-in functions, where type information is inferred from the types of operands and arguments. Once the AST has been annotated with type information, the LLL generation pass is greatly simplified. This makes both type checking and LLL generation easier to evaluate/audit.

As an aside, it's possible that separating type checking into its own compiler pass might allow the language/compiler's treatment of typing integer literals to change to be more context-dependent. That is, perhaps the language/compiler could treat an integer literal to be whatever type makes the most sense based on how it used rather than based on the literal itself. For instance, `7` could just as easily be an `int128` as a `uint256`, and that choice could be deferred until type checking.

6 Intermediate Representation (LLL)

The Vyper compiler adapts the LLL intermediate representation for its use. The LLL does not appear to have a specification, which places an extraordinary burden on anybody trying

to understand the IR generation phase.

Similarly, without a precise LLL spec, it is impossible to audit the LLL→bytecode pass of the compiler.

7 Needed Specifications

Before a full security audit of the Vyper compiler is performed, detailed specifications should be written.

7.1 Language Specification

The Vyper language lacks a complete, precise specification, which makes it impossible to know if the implementation conforms to the desired behavior. A language for implementing smart contracts must be well-specified so that programmers can be confident that they know what their smart contracts will actually do.

A specification for Vyper must include a complete description of the syntax, the semantics, the types, the statements, the operations, etc., of the language. Those descriptions must be sufficiently detailed that a conforming compiler could be written from the specification alone, without needing to consult the implementation of the reference compiler. (It is impossible to check a compiler's correctness if the definition of the language is based on *that* compiler's implementation and no specification.)

One test of the completeness of a language specification is to go through the compiler and determine if every action it takes can be traced back to something in the specification. If not, then either the specification is incomplete, or the compiler is doing something it shouldn't.

7.2 Interface Specifications

To independently examine the phases of a compiler, it is necessary to understand their interfaces precisely. In the Vyper compiler, this means that the AST definition, the type system's classes, and the LLL require thorough documentation. For each class representing those interfaces, each field must include a specification that indicates (1) the type of that field, (2) whether the field can be null, and (3) any non-type-based restrictions on the values that the field might hold. Extra credit for any invariants that can be added.

8 Bytecode Verification

Well-formed bytecode has properties than can be checked statically in much the same way that statically-typed programs can be checked for type correctness. For example, at any given instruction in a well-formed EVM program, the evaluation stack should have exactly the same height no matter what **dynamic** execution path was followed to reach that instruction. If the code generated by the compiler does not have this property, then something is almost certainly amiss.

Java popularized the requirement that bytecode be automatically verified before it would be trusted for execution. Java's bytecode verifier tested not only the stack height property described above, but it also tracked the types of values on the stack and made sure that (1) the types on the stack were consistent independent of dynamic execution path, and (2) that the types were not used incorrectly. For example, the verifier would prove statically that no execution path would add two object references together, or assign an integer value to a location that expected an object reference. For more details, see https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.10

While the EVM does not require that bytecode pass any static verification tests, such a tool would be invaluable to building confidence that the Vyper compiler was emitting reasonable bytecode. While it may not make sense to go all the way to implementing a tool that can check type safety, it would be good to have confidence that the stack heights are behaving consistently.

8.1 "Valency" in the code generator

The AST→LLL and LLL→EVM passes do some subtle/complex operations to make sure that the stack heights do not get out of sync—a property that a verifier could check.

We believe that some of the complexity is unnecessary because it seems to stem from the use of `if` in LLL to both translate statements (that should leave the stack empty), and expressions (that should add value(s) to the stack). This gets complicated by the fact that not all `if` s have an `else`, which means special care should be taken when the `if` is an expression. We would recommend introducing an `if-expr` operator to LLL and then asserting that all `if` s leave the stack empty (on both paths), and that all `if-expr` have both a `then` and `else` part, and that both add the same number of values to the stack.

9 Auditable Code

Just as the compiler could benefit from being decomposed into a number of compiler phases, the code itself could benefit from refactoring. Small functions with well-defined interfaces can be analyzed independently.

9.1 Refactor Long Functions

Long functions are hard to read and often indicate that a function is doing several things at once. For example, `arithmetic()` in `parser/expr.py` is over 250 lines of code. This function could be broken up into multiple functions, each handling a single binary operation. There are many such examples in the code, particularly in `parser/expr.py` and `parser/stmt.py`.

9.2 Use Static Typing

Most of the Vyper compiler is lacking type annotations or has incorrect annotations. For example, `dict_to_ast()` seems to have an incorrect type annotation. The parameter is annotated with the type `dict`, but the body of the function is clearly meant to handle multiple different types:

```
def dict_to_ast(ast_struct: dict) -> vyper_ast.VyperNode:
    if isinstance(ast_struct, dict) and 'ast_type' in ast_struct:
        ...
    elif isinstance(ast_struct, list):
        ...
    elif ast_struct is None or isinstance(ast_struct, (str, int)):
        ...
    else:
        ...
```

We recommend using type annotations everywhere. Not only will this aid readability considerably, but it will likely catch bugs that could otherwise only be found with comprehensive testing.

9.3 Avoid Code Repetition

Code repetition increases the burden on an auditor or other code reader. They must examine similar code multiple times instead of reading a reusable function a single time.

Code repetition also increases the chance of introducing subtle bugs. As an example, `concat()` in `functions/functions.py` and `list_literals()` in `parser/expr.py` both check that a series of values have the same type. They do this in subtly different ways, and the version in `list_literals()` has a bug (included at the end of this report).

Other examples of code repetition are type checking and "clamping" integer values to an allowed range. Similar variants of these are spread throughout the codebase.

10 Issues

Below is a list of issues discovered during the security review. It is by no means comprehensive, as finding bugs was not the primary goal of the review.

10.1 `if` / `else` stack height mismatch honeypot

This issue is the same as https://github.com/ethereum/vyper/issues/1511 but shows how it can be used for nefarious purposes.

It appears in the below code that someone can call the default function with enough to get over the 1 ether threshold and then call the default function multiple times with no ether to drain the contract. (`withdraw()` fails to clear the account's balance and can thus be called repeatedly.)

However, it's impossible to reach the `withdraw()` function because when the `if` branch is not taken in `__default__()`, a stack underflow occurs, and the transaction is reverted.

The contract owner can retrieve all the locked funds after the one-week expiration has elapsed.

```
owner: public(address)
balances: public(map(address, uint256(wei)))
expiration: public(timestamp)

@public
def __init__():
    self.owner = msg.sender
    self.expiration = block.timestamp + (60 * 60 * 24 * 7)

@public
```

```
@payable
def __default__():
    if msg.value > 0:
        assert msg.value >= as_wei_value(1, "ether")
        self.deposit(msg.sender, msg.value)

    if msg.value == 0:
        self.withdraw(msg.sender)

@public
def kill():
    assert block.timestamp > self.expiration
    selfdestruct(self.owner)

@private
def deposit(account: address, amount: uint256(wei)) -> uint256(wei):
    self.balances[account] += amount
    return self.balances[account]

@private
def withdraw(account: address):
    send(account, self.balances[account])
```

10.2 `if` / `else` branches with different resulting stack height can lead to stack manipulation

This is related to https://github.com/ethereum/vyper/issues/1511 but explores the more general problem:

```
@private
@constant
def foo() -> uint256:
    return 42

@private
@constant
def echo(a: uint256) -> uint256:
    if False:
```

```
        self.foo()
    return a

@public
@constant
def test() -> uint256:
    a: uint256 = 123
    if False:
        pass
    else:
        self.foo()
    self.echo(987)
    return a # BUG: returns 384
```

10.3 `uint256` exponentiation can overflow without error as long as the result is greater than the base

```
@public
@constant
def test(x: uint256, y: uint256) -> uint256:
    return x**y # BUG: test(10, 78) returns 736632861014704366114321199304967
```

This issue is due to the following code:

code/vyper/parser/expr.py:L646-L658

```
if ltyp == rtyp == 'uint256':
    o = LLLnode.from_list([
        'seq',
        [
            'assert',
            [
                'or',
                ['or', ['eq', right, 1], ['iszero', right]],
                ['lt', left, ['exp', left, right]]
            ],
        ],
```

```
        ['exp', left, right],
    ], typ=BaseType('uint256'), pos=getpos(self.expr))
```

10.4 `int128` exponentiation can overflow without error

```
@public
@constant
def test(x: int128, y: int128) -> int128:
    return x**y # BUG: test(2, 256) returns 0
```

The relevant compiler code is here:

code/vyper/parser/expr.py:L659-L667

```
elif ltyp == rtyp == 'int128':
    new_unit = left.typ.unit
    if left.typ.unit and not isinstance(self.expr.right, ast.Name):
        new_unit = {left.typ.unit.copy().popitem()[0]: self.expr.right.n}
    o = LLLnode.from_list(
        ['exp', left, right],
        typ=BaseType('int128', new_unit),
        pos=getpos(self.expr),
    )
```

10.5 Struct literals are dependent on key order

Note that the same values for `a` and `b` are set in both test functions, just in a different order:

```
struct Foo:
    a: uint256
    b: uint256

@public
@constant
def test() -> uint256:
```

```
    foo: Foo = Foo({a: 1, b: 2})
    return foo.a # returns 1

@public
@constant
def test2() -> uint256:
    foo: Foo = Foo({b: 2, a: 1})
    return foo.a # BUG: returns 2
```

This issue is due to the following code, which is dependent on the order keys are returned:

code/vyper/parser/expr.py:L1107-L1111

```
return LLLnode.from_list(
    ["multi"] + [o[key] for key in (list(o.keys()))],
    typ=StructType(members, name, is_literal=True),
    pos=getpos(expr),
)
```

Note that the behavior may differ depending on what version of Python is used to run the compiler.

10.6 Private functions can't have duplicate function selectors

This is an unnecessary constraint. Label names can be made unique.

```
@private
@constant
def gfah(): pass

@private
@constant
def eexo(): pass

# error: Label with name priv_236395036 already exists!
```

10.7 Event packing leaks a variable into the current context

```
Foo: event({a: bytes[25]})

@public
@constant
def test() -> string[7]:
    log.Foo("testing")
    return _log_pack_var_6_12 # returns "testing"
```

10.8 Inappropriate overflow check for `addmod` and `mulmod`

`addmod` and `mulmod` should allow arbitrary `uint256` operands, but the following code performs a check for addition or multiplication overflow:

code/vyper/functions/functions.py:L1023-L1052

```
@signature('uint256', 'uint256', 'uint256')
def uint256_addmod(expr, args, kwargs, context):
    return LLLnode.from_list(
        [
            'seq',
            ['assert', args[2]],
            ['assert', ['or', ['iszero', args[1]], ['gt', ['add', args[0], arg
            ['addmod', args[0], args[1], args[2]],
        ],
        typ=BaseType('uint256'),
        pos=getpos(expr),
    )


@signature('uint256', 'uint256', 'uint256')
def uint256_mulmod(expr, args, kwargs, context):
    return LLLnode.from_list(
        [
            'seq',
            ['assert', args[2]],
            ['assert', [
                'or',
                ['iszero', args[0]],
```

```
                ['eq', ['div', ['mul', args[0], args[1]], args[0]], args[1]],
            ]],
            ['mulmod', args[0], args[1], args[2]],
        ],
        typ=BaseType('uint256'),
        pos=getpos(expr),
    )
```

This results to the following runtime errors:

```
@public
@constant
def test1(a: uint256, b: uint256) -> uint256:
    return uint256_addmod(a, b, 12) # test(2**255, 2**255) reverts

@public
@constant
def test2(a: uint256, b: uint256) -> uint256:
    return uint256_mulmod(a, b, 12) # test(2**255, 2) reverts
```

10.9 Min and max can incorrectly use signed comparisons depending on argument order

```
@public
@constant
def test1() -> uint256:
    return min(0, 2**255) # returns 2**255

@public
@constant
def test2() -> uint256:
    return min(2**255, 0) # returns 0
```

This is due to the following code:

code/vyper/functions/functions.py:L1174-L1177

```
if left.typ.typ == 'uint256':
    comparator = 'gt' if is_min else 'lt'
else:
    comparator = 'sgt' if is_min else 'slt'
```

10.10 Possible simplification for list return type checking

It seems that the following check:

code/vyper/parser/stmt.py:L869-L877

```
sub_base_type = re.split(r'\(|\[', str(sub.typ.subtype))[0]
ret_base_type = re.split(r'\(|\[', str(self.context.return_type.subtype))[0]
loop_memory_position = self.context.new_placeholder(typ=BaseType('uint256'))
if sub_base_type != ret_base_type:
    raise TypeMismatchException(
        f"List return type {sub_base_type} does not match specified "
        f"return type, expecting {ret_base_type}",
        self.stmt
    )
```

can simply be:

```
if sub.typ.subtype != self.context.return_type.subtype:
    raise TypeMismatchException(
        f"List return type {sub.typ.subtype} does not match specified "
        f"return type, expecting {self.context.return_type.subtype}",
        self.stmt
    )
```

10.11 List literals can have mismatched types due to an off-by-one error

This check should use `> 0` instead of `> 1`:

code/vyper/parser/expr.py:L1066-L1067

```
if len(o) > 1 and previous_type != current_type:
    raise TypeMismatchException("Lists may only contain one type", self.expr)
```

The result is inconsistent type checking for list literals:

```
@public
def test():
    x: uint256[3] = [2**255, 1, 3] # Compiles without error
    y: uint256[3] = [1, 2**255, 3] # Error: Lists may only contain one type
```

10.12 Modulo at compile time differs from modulo at runtime

At compile time, constant expressions are computed using Python's built-in modulo operator. It handles negative numbers differently from the EVM:

```
@public
@constant
def foo() -> int128:
    return -5%2 # returns 1

@public
@constant
def bar(n: int128) -> int128:
    return n%2 # bar(-5) returns -1
```

10.13 Negative constants can be used in some places where `uint256` is required

The following code casts `int128` literals to `uint256` without any check on the value:

code/vyper/parser/parser_utils.py:L415-L417

```
# Integer literal conversion.
elif (frm.typ, to.typ, frm.is_literal) == ('int128', 'uint256', True):
    return LLLnode(orig.value, orig.args, typ=to, add_gas_estimate=orig.add_ga
```

This permits the following (invalid) code to compile without error:

```
Test: event({ n: uint256 })
x: map(uint256, uint256)

@public
def test():
    a: uint256 = self.x[-7]
    log.Test(-7)
```

10.14 Private function calls with arguments needlessly store a function selector in memory

The following Vyper code:

```
@private
@constant
def bar(a: uint256):
    pass

@public
def foo():
    self.bar(1)
```

produces the following IR:

```
...
/* Internal Call: bar */
[pop,
  [seq_unchecked,
    [seq, [mstore, 320, 69443890], [mstore, 352, 1]],
...
```

Note that `69443890 == 0x423a132 == bytes4(keccak256("bar(uint256)"))`.

Storing this in memory doesn't seem to have any purpose.

10.15 Runtime error when making an external call to the same contract

This code makes it an error to make an external call to the same contract:

code/vyper/parser/external_call.py:L75

```
['assert', ['ne', 'address', contract_address]],
```

This is a surprising limitation. It doesn't seem to have a clear benefit, and it could be problematic. As an example, multisig wallets often use self-calls to perform administrative functions. This way the wallet owners have to agree to make a change like lowering the required threshold of signatures. In Vyper, this would produce a runtime error.

10.16 `ZERO_WEI` constant can be defined with no effect

`ZERO_WEI` is a built-in constant, so it shouldn't be allowed as a user-defined constant:

```
ZERO_WEI: constant(uint256(wei)) = 3
ONE_WEI: constant(uint256(wei)) = 5

@public
@constant
def zero() -> uint256(wei):
    return ZERO_WEI # returns 0, not 3

@public
@constant
def one() -> uint256(wei):
    return ONE_WEI # returns 5, as expected
```

10.17 Decimals can't be the base for exponentiation

This might be an intentional restriction, but the error messages are confusing:

```
@public
def test(x: decimal) -> decimal:
```

```
    return x**convert(2, decimal) # Error: Only whole number exponents are sup
    return x**2 # Error: Cannot implicitly convert decimal to int128.
```

10.18 Array types are not hashed for event topics

Solidity hashes all array types, including `bytes` and `string`s, when they're used as indexed parameters (event topics).

Vyper only accepts `bytes` and `string`s with a maximum length of 32. They're then converted to `bytes32`. This could be an interoperability problem. Vyper doesn't appear to accept other array types as indexed event parameters at all.

10.19 Incorrect error message when converting a long bytes array

The following code results in the error message `Cannot convert bytes array of max length 320 to uint256`:

```
@public
@constant
def test(b: bytes[34]) -> uint256:
    return convert(b, uint256)
```

This is a typo in the error message here:

code/vyper/types/convert.py:L229

```
f"Cannot convert bytes array of max length {in_arg.value} to uint256",
```

The error should use `in_arg.typ.max_len` instead. Note that this same mistake occurs multiple places in the code.

10.20 Code that ends in a comment (with no newline) cannot be compiled

This was previously reported for Vyper, but it's actually a Python bug:

```
@public
def __init__():
   pass

# BUG (no newline)
```

10.21 Struct getter collisions

Due to the way getter names are generated for struct fields, collisions can happen:

```
struct Test:
    foo: uint256
    __foo: uint256

a: public(Test)
a__: public(Test)
```

The error message is `Duplicate function name: a____foo`. This is better than an ambiguity, but perhaps the collision should be avoided altogether by using a separator that cannot appear in identifiers.

The following code is responsible for this naming:

code/vyper/parser/global_context.py:L230-L240

```
# Struct type: for each member variable, make a separate getter, extend
# its function name with the name of the variable, do not add input
# arguments, add a member access to the return statement
elif isinstance(typ, StructType):
    o = []
    for k, v in typ.members.items():
        for funname, head, tail, base in cls._mk_getter_helper(v, depth):
            o.append(("__" + k + funname, head, "." + k + tail, base))
    return o
else:
    raise Exception("Unexpected type")
```

10.22 Relative imports are broken beyond one parent level

The following code produces paths like `..../something.vy` instead of `../../../something.vy`:

code/vyper/signatures/interface.py:L225-L226

```
if level:
    base_path = f"{'.'*level}/{module.replace('.','/')}"
```

A path like `..../something.vy` is actually valid, so this can end up importing code from a surprising place. The expected behavior is undocumented, so perhaps the intention was just to disallow more than two leading dots.

10.23 Exponentiation of a type with units works differently for `uint256` and `int128`

When the underlying type is `uint256`, all units are stripped away. When the underlying type is `int128`, the units remain but with an incorrect exponent (`m**3` rather than `m**6` below):

```
units: {
    m: "meter"
}

@public
@constant
def test() -> int128(m**3):
    a: int128(m**2) = 5
    return a**3 # BUG: Type int128(m**3) instead of int128(m**6)

@public
@constant
def test2() -> uint256:
    a: uint256(m**2) = 5
    return a**2 # BUG: Type uint256 instead of uint256(m**4)
```

10.24 Negation (“unary subtraction”) can underflow

There is no check for underflow when negating:

```
@constant
@public
def bar(x: uint256) -> uint256:
    return -x # BUG: returns 2**256 - x
```

Using `0 - x` instead reverts as expected.

The issue is here, where no check is done for underflow:

code/vyper/parser/expr.py:L969-L985

```
elif isinstance(self.expr.op, ast.USub):
    if not is_numeric_type(operand.typ):
        raise TypeMismatchException(
            f"Unsupported type for negation: {operand.typ}",
            operand,
        )

    if operand.typ.is_literal and 'int' in operand.typ.typ:
        num = ast.Num(n=0 - operand.value)
        num.source_code = self.expr.source_code
        num.lineno = self.expr.lineno
        num.col_offset = self.expr.col_offset
        num.end_lineno = self.expr.end_lineno
        num.end_col_offset = self.expr.end_col_offset
        return Expr.parse_value_expr(num, self.context)

    return LLLnode.from_list(["sub", 0, operand], typ=operand.typ, pos=getpos(
```

10.25 Type confusion with timestamp negation

Subtracting two timestamps yields a `timedelta`, but `-x` where `x` is a timestamp is still a `timestamp`:

```
@public
def test(x: timestamp) -> timestamp:
    return -x
```

Note that this negation probably shouldn't be allowed at all, given that `timestamp` is an unsigned value, so perhaps this issue will go away when that is fixed.

10.26 Array being iterated over can be modified in another function

Modifying the array directly in the loop body is disallowed, so this could be considered a bug:

```
x: uint256[1]

@private
def store_in_x(index: uint256, value: uint256):
    self.x[index] = value

@public
def foo():
    for n in self.x:
        # self.x[0] = n * 2 # blocked with "Altering list 'self.x' which is be
        self.store_in_x(0, n * 2) # BUG: allowed
```

10.27 Infinite loop via modifying loop index

The generated variable `_index_for_*` shouldn't be accessible. The following generates an infinite loop:

```
@public
def infinite_loop():
    for n in [1,2,3]:
        _index_for_n = 0 # BUG
```

10.28 Can't loop over a nested array

`for` loops in Vyper can loop over arrays, but not if they're nested. This code fails to compile with the surprising error `'Subscript' object has no attribute 'func'`:

```
@public
def __init__():
    x: uint256[5][2] = [[0, 1, 2, 3, 4], [2, 4, 6, 8, 10]]
    for i in x[1]:
        pass
```

10.29 State variables can be named `public` and `constant`

These should be treated as reserved words.

```
public: uint256 # BUG
constant: uint256 # BUG

@public
def __init__():
    self.public = 5
```

10.30 `else` allowed on `for` loop silently ignored

The following code compiles, but the `else` is ignored. (No code is generated for it.)

```
x: public(uint256)

@public
def __init__():
    for i in range(10):
        self.x += 1
    else:
        self.x = 123
```

10.31 Circumventing `@constant` with multiple return values and tuple assignment

Modifying state is not allowed in `@constant` functions, but you can bypass the check by using multiple return values:

```
x: public(uint256)

@private
@constant
def returnTwo() -> (uint256, uint256):
    return 1, 2

@public
@constant
def foo():
    a: uint256 = 0
    a, self.x = self.returnTwo() # BUG
```

Appendix 1 - Disclosure

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.