PUBLIC

Limited Code Review

of the Vyper Compiler IR optimizer and safe-math module

July 22, 2022

Produced for

by

Contents

1 Executive Summary

Dear Vyper team,

Thank you for trusting us to help Vyper with this review. Our executive summary provides an overview of subjects covered in our review of the latest reviewed contracts of Vyper Compiler according to Scope to support you in forming an opinion on their security risks.

Limited code reviews are best-effort checks and don't provide assurance comparable to a non-limited code review. This review was not conducted as an exhaustive search for bugs, but rather as a best-effort sanity check for files of interest. The review was executed by one engineer over a period of two weeks supported by a second engineer for four days. Given the large scope and codebase and the limited time, the findings aren't exhaustive.

Vyper implements a compiler of Vyper language into EVM bytecode.

The most critical subjects covered in our review are the functional correctness of arithmetic operations and the soundness of performed optimizations. Security regarding functional correctness of arithmetic operations is improvable, due to discovered bugs, where IR nodes introduced by safemath, can themselves have overflows.

We did not uncover any issues regarding the soundness of performed optimizations, however, we would like to note that current optimizations are applicable only in a very limited number of cases. Extending the applicable cases when they can be applied might lead to potential problems and bugs. In addition, since optimizations are performed after safemath, extending optimizations to smaller than 256-bit datatypes should be done carefully. Some of the currently performed optimizations might potentially lead to an overflow of smaller datatypes, if not properly adjusted.

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	7

2 Review Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The review was performed on the following source code files inside the `https://github.com/vyperlang/vyper` repository:

- `vyper/ir/optimizer.py`. Functions inside this file were checked, with a special focus on `_optimize_binop`.
- `vyper/codegen/expr.py`. Functional correctness of `parse_BinOp` function was checked.
- `vyper/codegen/arithmetic.py`. Functions inside this file were checked for functional correctness.

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	23 June 2022	8eed2579f69b3081818423c0260696e397696361	Initial Version

2.1.1 Excluded from scope

All other files and imports that were not mentioned in Scope.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Review Overview.

Vyper language is a pythonic smart contract oriented language, targeted at Ethereum Virtual Machine (EVM). Vyper compiler translates the Vyper language into the EVM bytecode. The compilation process is performed in multiple phases:

1. Vyper AST is generated from the Vyper source code.
2. Constant folding is performed on the AST.
3. Imported interfaces are added to produce the contextualized global AST.
4. The Intermediate Representation (IR) is produced from the contextualized AST.
5. Optimizations are performed on the IR.
6. Assembly is generated from the IR.
7. Bytecode is generated from the Assembly.

In this review, we focused on 2 parts:

1. The checks for overflows, that are introduced for arithmetic operations during the IR generation.
2. The optimizations that are performed on IR.

2.2.1 Arithmetic checks

The EVM opcodes like `ADD`, `MUL` or `POW` do not revert on overflows. The Vyper corresponding ops `+`, `*` and `**` perform overflow checks and revert on overflow by default. These checks are introduced in the IR during the conversion of the contextualized AST. In addition, most EVM opcodes operate with 32-byte words. In Vyper, N-bit signed and unsigned integers as well as a decimal type are available. The IR safemath checks the need to set the correct boundaries for each data type so that the result of the operations is not only within the bounds of the word but also of the given type.

2.2.2 IR optimizations

Various optimizations are defined in `vyper/ir/optimizer.py` file. They are performed numerous times on each node of the IR, until they stop changing. The following optimizations are implemented:

1. Binary operator: If one or both operands are literals whose value are known at compile time, some optimizations can be introduced, for example `x+0` can be safely replaced by `x`.
2. Unary operator: In some cases, unary operators such as `slr`, `ceil32` or `iszero` can be optimized if the operand is a literal with some given value. `iszero(0)` would be reduced to `1` for example.
3. Unary operator: In some cases, unary operators such as `slr`, `ceil32`, `assert` or `iszero` can be optimized. `iszero(0)` would be reduced to `1` for example.
4. Branches of an `if` node can be either removed if the condition is known at compile time, either swapped to allow for further optimizations in some cases.
5. `mzero` and `calldatacopy` operations that are zeroing memory can be merged into a single `calldatacopy`.
6. Empty sequences can be removed from the Intermediate Representation.
7. Sequential operations copying from calldata to memory can be merged into a single `calldatacopy` operation.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

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

- 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	7

5.1 Compiler Panicking for an Out of Range Integer Node

Correctness Low Version 1

CS-VYPER_JULY_2022-001

For exponentiation of unsigned numbers, the following IR node is added to check for overflows of the computation:

```
["lt", x, upper_bound]
```

When a `uint256` is raised to the power of 0 or 1, `upper_bound` is equal to `MAX_UINT256+1`, which is an output of `calculate_largest_base`. When trying to create a node for such value, the `__init__` function of `IRnode`, will throw an exception as `-(2 ** 255) <= self.value < 2 ** 256` is false.

```
@external
def foo() -> uint256:
    x: uint256 = 0
    return x ** 1
```

```
Error compiling: Foo.vy
vyper.exceptions.CompilerPanic: out of range

This is an unhandled internal compiler error. Please create an issue on Github to notify the developers.
https://github.com/vyperlang/vyper/issues/new?template=bug.md
```

5.2 Different Semantics for Raising to the Power of Negative Numbers

Correctness Low Version 1

CS-VYPER_JULY_2022-002

When computing the power of 0 or 1 by a negative number, if the compiler knows the exponent at compile time, it will output an exception. On the other side, if the compiler is only aware of the base, it will successfully compile and running the code will not revert. This behavior is due to the fact that the runtime checks added by the compiler only check that the result of the computation is in bounds. It does not check that the exponent is not negative to have the same semantic as the other case. The two examples below show the issue, compilation of `foo` will output the given exception while a call to `bar` will return 1.

```
@external
def foo() -> int16:
    x: int16 = 1
    return x ** (-2)
```

```
vyper.exceptions.InvalidOperation: Cannot calculate a negative power
contract "Foo.vy", function "foo", line 4:17
     3     x: int16 = 1
---> 4     return x ** (-2)
-------------------------^
     5
```

```
@external
def bar() -> int16:
    x: int16 = -2
    return 1 ** x
```

5.3 Exponentiation Lead to `CompilerPanic` Exception

Correctness Low Version 1

CS-VYPER_JULY_2022-003

In `arithmetic.safe_pow`, when the base `x` is a literal, either `x in (0,1)`, either `calculate_largest_power` is called. In `calculate_largest_power` however, `CompilerPanic` is raised when the absolute value of the base is either 0 or 1. This behavior results in powers of `-1` raising the exception as its absolute value is 1.

```
a = abs(a)  # No longer need to know if it's signed or not
if a in (0, 1):
        raise CompilerPanic("Exponential operation is useless!")
```

The following code snippet produces this behavior:

```
@external
def foo():
        x: int256 = 4
        y: int256 = (-1) ** x
```

In addition the compiler does not handle the exception that is produced by this snippet:

```
Error compiling: Foo.vy
vyper.exceptions.CompilerPanic: Exponential operation is useless!

This is an unhandled internal compiler error. Please create an issue on Github to notify the developers.
https://github.com/vyperlang/vyper/issues/new?template=bug.md
```

5.4 Folding Does Not Follow the Vyper Runtime Semantics

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

CS-VYPER_JULY_2022-004

AST folding is performed at the very beginning of the compilation pipeline. As it happens before the semantics validation, depending on the expression, it is possible that either the folding is too restrictive compared to the real semantics, or an expression that is not supposed to be valid is folded:

The following example shows the first case:

```
@external
def foo1() -> int8:
    return 1 ** (-5)
```

```
@external
def foo2() -> int8:
    x: int8 = (-5)
    return 1 ** x
```

Compiling `foo1` outputs the following exception since folding does not allow exponents to be negative. On the other side, calling `foo2` returns `1` as no folding is happening on the exponentiation.

```
Error compiling: Foo1.vy
vyper.exceptions.InvalidOperation: Cannot calculate a negative power
contract "Foo1.vy", function "foo", line 3:11
     2 def foo1() -> int8:
---> 3     return 1 ** (-5)
------------------^
     4
```

This example shows how some expression are folded even if they should not be valid:

```
@external
def bar1() -> uint16:
    return 1 - 2 + 2
```

```
@external
def bar2() -> uint16:
    x: uint16 = 2
    return 1 - x + 2
```

In this example, calling `bar1` returns 1 while calling `bar2` reverts since no folding is done and `1-x` results in an underflow.

5.5 SafeMath Reverting on Valid Power

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

CS-VYPER_JULY_2022-005

When having the exponentiation of a signed number with the exponent `y` being a literal, `safe_pow` adds the following check to the intermediate representation:

```
ok = ["and", ["slt", x, upper_bound], ["sgt", x, -upper_bound]]
```

Signed integers are represented using two's complement, one of the properties of this representation is that `MIN_INT+1 == -MAX_INT`.

For values of `y` for which there exists `x'` such that `x'**y==MAX_INT+1`, `safe_pow` will compute `upper_bound=x'`, which is result of `calculate_largest_power`. If the event that the base of the exponentiation happens to be `x==-x'`, although there is no overflow as `x**y==MIN_INT`, the check mentioned above will fail as `["sgt", x, -upper_bound]` will return false since `x==-upper_bound`.

For instance, when the following code snippet is compiled and deployed, a call to `foo` will revert:

```
@external
def foo() -> int16:
        x: int16 = -8
        y: int16 = x ** 5
return y
```

The produced IR check will be:

```
[seq, [assert, [and, [slt, x, 8], [sgt, x, -8]]], [exp, x, 5 <5>]]]]],
```

Note that the same issue arises when trying to compute `MIN_INT**0` as well since `upper_bound==MAX_INT+1` in this case:

```
@external
def foo() -> int16:
        x: int16 = -32768
        y: int16 = x ** 0
return y
```

5.6 SafeMath Reverting on Valid Power for `int256`

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

When having an exponentiation of a signed number, the following IR node is added to check for overflows and underflows of the computation:

```
["and", ["slt", x, upper_bound], ["sgt", x, -upper_bound]]
```

For `int256`, a similar issue as the one described in SafeMath Reverting on Valid Power can happen. For any exponentiation of a `int256` by 0 or 1, `upper_bound` will be equal to `MAX_INT256+1` and hence the left-hand side of the `and` will be the following node: `N=["slt", x, MAX_INT256+1]`. This check will always evaluate to false as the EVM interprets `MAX_INT256+1` as `MIN_INT256`. In the following code, a call to `foo` will always revert:

```
@external
def foo() -> int256:
    x: int256 = 2
    return x ** 0
```

When optimizations are enabled an interesting case can happen, during its call to `_comparison_helper`, the optimizer will check the following:

```
if is_strict and _int(args[1]) == never:
    # e.g. gt x MAX_UINT256, slt x MIN_INT256
    return (0, [])
```

As `_int(MAX_INT256+1)` evaluates to `MIN_INT256`, `N` will be replaced by the integer node `0`.

5.7 Wrong Overflow Exception

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

CS-VYPER_JULY_2022-007

When having an exponentiation, if the exponent `y` is a literal, `safe_pow`'s call to `calculate_largest_base` will raise a `TypeCheckFailure` if `y` happens to be greater than the number of bits the given type can store. While this is a correct behavior in most cases, for bases equal to 0 or 1, such computation would not overflow.

In practice, an `OverflowException` is raised earlier when the same check is performed against the Vyper AST in `validate_numeric_op`.

```
@external
def foo():
        x: uint256 = 1
        y: uint256 = x ** 257
```

```
Error compiling: Foo.vy
vyper.exceptions.OverflowException: Power is too large, the calculation will always overflow
contract "test_opti/02.vy", function "foo", line 4:22
     3     x: uint256 = 1
---> 4     y: uint256 = x ** 257
-----------------------------^
     5
```