PUBLIC

Limited Code Review

of the Vyper Compiler

Semantic analysis and Code generation

September 19, 2023

Produced for

by

Contents

1 Executive Summary

Dear Vyper team,

Thank you for trusting us to help Vyper with this review. Our executive summary provides an overview of subjects covered in our review of the latest version of Vyper Compiler according to Scope to support you in forming an opinion on their security risks. The review was executed by one engineer over two weeks. It's important to note that, due to the extensive scope and codebase, our time-limited review does not capture the full depth of a comprehensive security analysis.

The subjects covered by our review are detailed in the Review Overview section.

We found that the O(1) selector table is a good optimization and provides substantial gas savings, especially for large contracts. As pointed out by Incorrect dense selector when one bucket is empty, this new feature brings some edge cases that are hard to cover with tests, and even using fuzzing. We recommend that testing should be performed with special care for such part of the compiler.

As described in Arguments buffer size too large when calling ecmul and ecrecover can return undefined data in some edge case, issues were found in the fixes of the recent security advisory. These issues were shortly fixed and we can confidently assert that the security advisories that were initially in scope for this review have been resolved.

The large number of issues found in the builtins functions shows that special attention should be given to this part of the compiler and more testing should be done on that side.

Finally, although Vyper v0.3.10 fixes a substantial amount of issue and improve the compiler greatly, the large number of high-severity issues discovered during this assessment along with the limited scope of this review make further assessments necessary.

The following sections will give an overview of the system and the issues uncovered. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	3
Medium-Severity Findings	2
Low-Severity Findings	9

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 commits and pull requests composing v0.3.10rc3 of the `https://github.com/vyperlang/vyper` repository:

The following commits were in scope:

https://github.com/vyperlang/vyper/compare/v0.3.9...v0.3.10rc3

Specifically, special attention was given to the following pull requests:

1. https://github.com/vyperlang/vyper/pull/3584
2. https://github.com/vyperlang/vyper/pull/3496
3. https://github.com/vyperlang/vyper/pull/3569
4. https://github.com/vyperlang/vyper/pull/3586
5. https://github.com/vyperlang/vyper/pull/3583
6. https://github.com/vyperlang/vyper/pull/3587

In addition to the pull requests mentioned, a non-exhaustive general review of `vyper.builtins` was conducted within the available time constraints.

This review was not conducted as an exhaustive search for bugs, but rather as a best-effort sanity check. The issues already documented on the `https://github.com/vyperlang/vyper` repository at the time of the review were not included in this report.

2.2 System Overview

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

1. Vyper Abstract Syntax Tree (AST) is generated from Vyper source code.
2. Literal nodes in the AST are validated.
3. Constants are replaced in the AST with their value and constant expressions are folded.
4. Module-level objects are added to the namespace and top-level declarations are validated.
5. Getters for public variables are added, and unused statements are removed from the AST.
6. The semantics of the program are validated. The structure and the types of the program are checked and type annotations are added to the AST.
7. Positions in storage and code are allocated for storage and immutable variables.
8. The Vyper AST is turned into a lower-level intermediate representation language (IR).
9. Various optimizations are applied to the IR.
10. The IR is turned into EVM assembly.
11. Assembly is turned into bytecode, essentially resolving symbolic locations to concrete values.

We now give a brief overview of the two main components we are interested in: the semantic validation and the code generation.

2.2.1 Semantic Validation

Once the Abstract Syntax Tree has been folded, it is analyzed to ensure that it is a valid AST regarding Vyper semantics and annotated with types and various metadata so that the code generation module can properly generate IR nodes from the AST.

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.

Once all module-level statements have been properly analyzed, the compiler checks some properties of the module, for example, whether there are no circular function calls or collisions between function selectors.

At this point, getters for public variables are added to the AST and unused statements are removed from it, this includes for example constant variable declaration, which is no longer needed as the constant has been inlined and the declaration has been validated.

The `FunctionNodeVisitor` 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 analyze the type of the statement's sub-expressions.

The `_ExprAnalyser` is used 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 `FunctionNodeVisitor` 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.

Once the full Vyper contract has been validated, the `StatementAnnotationVisitor` and the `ExpressionAnnotationVisitor` are called to finally annotate the expression nodes of the AST with their corresponding type for the code generation that will happen later.

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, 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 `GlobalContext` containing the annotated AST as well as some properties such as variables and immutables information. 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.

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 codesize 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 Findings

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

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	3

- Incorrect Dense Selector When One Bucket Is Empty
- Incorrect Re-Entrancy Lock When Key Is Empty String
- Memory Corruption in Builtins Utilizing msize

Medium-Severity Findings	2

- _abi_decode Return Values Are Not Clamped
- ecrecover Can Return Undefined Data in Some Edge Case

Low-Severity Findings	9

- Arguments Buffer Size Too Large When Calling ecmul
- Arguments Passed to _abi_decode Are Not Type-Checked
- Builtins' varargs Are Not Properly Type-Checked
- Compiler Panics When Overflowing the Memory
- Documentation Inconsistencies
- Incorrect Typing of Literal Arrays When Used With len
- Outdated or Incorrect Comments
- Warning and Error Messages Inconsistencies
- extract32 Return Type Can Be Any byteM

5.1 Incorrect Dense Selector When One Bucket Is Empty

Correctness High Version 1

CS-VYPER_SEPTEMBER_2023-001

When generating a selector table with the `codesize` optimization, `_dense_jumptable_info()` tries to generate buckets and their corresponding magic number given a number of buckets. This function is called by `generate_dense_jumptable_info()` with different numbers of buckets to produce a table with as few buckets as possible. In the case where `_dense_jumptable_info()` is given some number of buckets to generate `n` but ends up with only `m` non-empty buckets (`m < n`), the returned python dict only contains `m` elements. As the code generation for the selector table uses `m` as the number of buckets, the latter will not behave as it should since:

- To get the bucket of a method id `x`, `x mod m` is computed at runtime although the function was placed in the bucket `x mod n` at compile time.
- The data section for the buckets header will miss one row for the empty bucket, meaning that even if there is some function with an id `x` such that `r = x mod m = x mod n`, if `r > b` where `b` was the of the empty bucket's id, the bucket `r - 1` will be searched instead of the bucket `r`.

As `m < n`, the bucket information (bucket data section, bucket magic number and bucket size) copied at runtime will always be some valid bucket and won't be some random data out of the header's data section. From there, the provided method ID is compared to the method ID of the function of the bucket whose magic number matches, the execution should revert as in the case where the wrong bucket is being used, no function will match the method ID. In other words, this issue makes functions of the contract revert when they should not, but does not allow for unexpected or restricted actions on the contract.

For example, calling the function `aJ3EJUYUK` on the following contract compiling with the `codesize` optimization will revert.

```
@external
def aJ3EJUYUK()->uint256:
    return 1
@external
def a9XXCI8PW()->uint256:
    return 2
@external
def aOBAJXNGO()->uint256:
    return 3
@external
def a0FUKH9AF()->uint256:
    return 4
@external
def a5ELEP7F4()->uint256:
    return 5
@external
def a3Y10PU1H()->uint256:
    return 6
@external
def aKX7ATQCX()->uint256:
    return 7
@external
def aT12B3E6Y()->uint256:
    return 8
@external
def a7U1B9OBW()->uint256:
    return 9
@external
def aGLOLGNUZ()->uint256:
    return 10
@external
def a7V3L4VPT()->uint256:
    return 11
@external
def aREJI588E()->uint256:
    return 12
@external
def a7XVZOF81()->uint256:
    return 13
```

```
@external
def aNJK5DHE9()->uint256:
    return 14
@external
def aDJT6R9PE()->uint256:
    return 15
```

5.2 Incorrect Re-Entrancy Lock When Key Is Empty String

Security High Version 1

CS-VYPER_SEPTEMBER_2023-002

The `nonreentrant` decorator does not protect functions in the case that the provided key is the empty string.

Note that this issue is being tracked in a security advisory [1]

```
@nonreentrant("") # unprotected
@external
def bar():
    pass

@nonreentrant("lock") # protected
@external
def foo():
    pass
```

5.3 Memory Corruption in Builtins Utilizing `msize`

Security High Version 1

CS-VYPER_SEPTEMBER_2023-003

As Vyper memory is statically allocated, in the cases where some data whose size is unknown at compile time must be stored in memory (calldata, `extcodecopy`), the compiler uses `msize` as the starting index of the memory that will be allocated for the data. In the general case, this solution is sufficient, however, in some edge cases, the compiler will allocate and use memory inside this buffer for other purposes after storing the data and before using them, thus, overwriting them.

The affected builtins are `raw_call`, `create_from_blueprint` and `create_copy_of`.

- For `raw_call`, the argument buffer of the call can be corrupted, leading to incorrect calldata in the sub-context. In this case, when the data to pass with the `raw_call` is `msg.data`, as there is no way to know the size of the calldata of the current context at compile time, the compiler uses `msize` as the beginning of the argument buffer for the call. As the argument `to` and the kwargs `value` and `gas` are evaluated after the `calldatacopy` and before the call, if their evaluation was to store to memory, it could overwrite the copied calldata.
- For `create_from_blueprint` and `create_copy_of`, the buffer for the to-be-deployed bytecode can be corrupted, leading to deploying incorrect bytecode. Again, the kwargs `value` and `salt` are evaluated after the `extcodecopy` or `codecopy` and before executing `create` or `create2`. If they were to store to memory, they could overwrite the buffer for the bytecode to be deployed.

Below are the conditions that must be fulfilled for the corruption to happen for each builtin:

5.3.1 raw_call

- The `data` argument of the builtin is `msg.data`.
- The `to`, `value` or `gas` passed to the builtin is some complex expression that results in writing to the memory.

5.3.2 create_copy_of

- The `value` or `salt` passed to the builtin is some complex expression that results in writing to the memory.

5.3.3 create_from_blueprint

- Either no constructor parameters are passed to the builtin or `raw_args` is set to True.
- The `value` or `salt` passed to the builtin is some complex expression that results in writing to the memory.

Note that more details and examples on the issue are described in the corresponding security advisory [2].

5.4 _abi_decode Return Values Are Not Clamped

Correctness **Medium** **Version 1**

CS-VYPER_SEPTEMBER_2023-004

The returned values of the builtin `_abi_decode` are not clamped and can be out of bound for the specified Vyper type.

The function `foo` of the contract below returns 3, meaning that `_abi_decode` returned `max_value(uint256)` which is out of bounds for `uint8` and that the addition overflowed 256 bits.

```
@external
def foo() -> uint8:
    x: Bytes[32] = _abi_encode(max_value(uint256), ensure_tuple = False)
    y: uint8 = _abi_decode(x, uint8, unwrap_tuple=False) + 4
    return y
```

5.5 ecrecover Can Return Undefined Data in Some Edge Case

Security **Medium** **Version 1**

CS-VYPER_SEPTEMBER_2023-005

This issue is an edge case of the recent security advisory [3] for `ecrecover` that was missed in the corresponding fix [4].

As the `ecrecover` precompile does not fill the output buffer (memory location 0) if the signature does not verify, the `ecrecover` builtin would return whatever was in the buffer before. This issue was fixed by cleaning any dirty bytes at memory location 0 before evaluating the builtin.

As the arguments of `ecrecover` are evaluated after this cleaning, if one of the arguments of the builtin was to be compiled to some bytecode storing to memory location 0, the data stored there would be the returned value of `ecrecover` in case of a signature that does not verify.

In the following contract, a `foo` would not revert as `owner` is stored at slot `0` by `get_v()`.

```
owner: immutable(address)

@external
def __init__():
    owner = 0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf

@internal
def get_v() -> uint256:
    assert owner == owner # force a dload to write at index 0 of memory
    return 21

@payable
@external
def foo():
    assert ecrecover(empty(bytes32), self.get_v(), 0, 0) == owner # True
```

5.6 Arguments Buffer Size Too Large When Calling `ecmul`

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

CS-VYPER_SEPTEMBER_2023-006

PR3583 [5] fixes an issue with the order of evaluation of the side effect of several builtins arguments [6]. Although the issue has been fixed with the PR, the builtin `ecmul` is performing a static call to the corresponding precompile with an argument buffer size too large (128 bytes instead of 96 bytes).

Note that this issue has been communicated and fixed before the PR was merged.

5.7 Arguments Passed to `_abi_decode` Are Not Type-Checked

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

CS-VYPER_SEPTEMBER_2023-007

The type of arguments passed to `_abi_decode` are not validated and ensured to match the expected argument types of the builtin.

```
# should not compile but compiles
@external
def bar(j:String[32]) -> bool:
    s:bool = _abi_decode(j, bool, unwrap_tuple= False)
    return s
```

```
# should not compile but the compiler panics with:
# AttributeError: 'IntegerT' object has no attribute 'maxlen'
```

```
@external
def foo(j:uint256) -> bool:
    s:bool = _abi_decode(j, bool, unwrap_tuple= False)
    return s
```

5.8 Builtins' `varargs` Are Not Properly Type-Checked

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

CS-VYPER_SEPTEMBER_2023-008

The builtins `create_from_blueprint`, `print` and `_abi_encode` use `varargs` to be able to accept an arbitrary number of additional arguments. The compiler only checks that these arguments have a proper Vyper type but does not enforce anything else. Passing for example an `Hashmap` or types in the `varargs` will make the compiler crash.

```
f:HashMap[uint256, uint256]
@external
def foo(blueprint: address):
    # vyper.exceptions.TypeCheckFailure: assigning HashMap[uint256, uint256] to HashMap[uint256, uint256] 128 0 <self.f>
    g:address =  create_from_blueprint(blueprint, self.f)
```

```
@external
def bar():
    # AttributeError: 'IntegerT' object has no attribute 'typ'
    print(uint16)
```

5.9 Compiler Panics When Overflowing the Memory

Design **Low** **Version 1**

CS-VYPER_SEPTEMBER_2023-009

Given some expression that overflows the memory, the compiler panics instead of exiting with some custom error.

For example, compiling the following contract results in `vyper.exceptions.CompilerPanic: out of range`.

```
@external
def bar():
    s:String[max_value(uint256)] = "a"
```

5.10 Documentation Inconsistencies

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

CS-VYPER_SEPTEMBER_2023-010

- In the documentation for `create_from_blueprint`, the keyword argument `raw_args` is not documented.
- `create_from_blueprint`'s example code snippet does not compile in the latest Vyper versions as the type `String` is used without maximum length.
- The documentation of `raw_call` mentions that in case `gas` is not set, all remaining gas is forwarded while in reality "all but one 64th" of the current contract's gas is forwarded.

5.11 Incorrect Typing of Literal Arrays When Used With `len`

Design Low Version 1

CS-VYPER_SEPTEMBER_2023-011

When trying to get the length of a literal array with the builtin `len`, the compiler fails with an `InvalidType` exception although literal arrays can usually be typed as dynamic arrays.

For example, the following contract fails to compile with an `InvalidType` exception.

```
@external
def foo():
    a: uint256 = len([1,2,3])
```

5.12 Outdated or Incorrect Comments

Correctness Low Version 1

CS-VYPER_SEPTEMBER_2023-012

The following comments are inconsistent with the compiler's behaviors.

- In `ModuleAnalyzer.__init__()`, "check for collisions between 4byte function selectors" is referring to a removed check.
- In `_MinMaxValue.evaluate()`, "TODO: to change to known_type once #3213 is merged" refers to a pull request that has been superseded and hence is outdated.
- The documentation for `generate_ir_for_external_function()` is not up to date as it mentions for example `check_nonpayable` which has been removed with the new selector table.
- The memory layout described in the comment of `CreateFromBlueprint._build_create_IR()` is incorrect.

5.13 Warning and Error Messages Inconsistencies

Correctness Low Version 1

CS-VYPER_SEPTEMBER_2023-013

The following warning and error messages could be improved.

- In `BitwiseNot.evaluate()`, the warning mention the bitwise xor operator ^ instead of ~.

- In `_validate_msg_data_attribute()`, the first raised exception do not mention that `msg.data` can be used with `raw_call`.

5.14 `extract32` Return Type Can Be Any `byteM`

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

CS-VYPER_SEPTEMBER_2023-014

According to both the documentation and the exception raised in `Extract32.infer_kwargs_types()`, `extract32` can have as return type `bytes32`, `address` or any `integer` type. However, given the check-in `Extract32.infer_kwargs_types`, it is possible to specify as return type any `bytesM`.

For example, the following code compiles:

```
@payable
@external
def foo(b:Bytes[32]):
    x: bytes8 = extract32(b, 0, output_type = bytes8)
```

1 https://github.com/vyperlang/vyper/security/advisories/GHSA-3hg2-r75x-g69m
2 https://github.com/vyperlang/vyper/security/advisories/GHSA-c647-pxm2-c52w
3 https://github.com/vyperlang/vyper/security/advisories/GHSA-f5x6-7qgp-jhf3
4 https://github.com/vyperlang/vyper/commit/019a37ab
5 https://github.com/vyperlang/vyper/pull/3583
6 https://github.com/vyperlang/vyper/security/advisories/GHSA-4hg4-9mf5-wxxq