Uniswap V2 in Vyper

Marvin Siu Yin Chan (mc5100)

mc5100@columbia.edu

1. Synopsis

1.1 Overview

The invention of the blockchain has allowed for a new computing paradigm - a programmable computer that can be used by anyone but is not owned by anyone - and code deployed on the blockchain is called a smart contract. This new computing paradigm has changed industries like finance and art in the form of decentralized finance and non-fungible tokens, respectively. However, since anyone can interact with any smart contract on the blockchain, security vulnerabilities in smart contracts can be exploited by malicious actors. For example, the DAO back in 2016 resulted in $60 million worth of assets stolen and required a hard fork to resolve the issue. More recently, the Wormhole hack resulted in $325 million worth of assets stolen. These hacks highlight the importance of security in smart contracts.

There are currently many blockchains like Ethereum, Solana, etc. that support smart contracts. Ethereum is currently the most popular blockchain for smart contract deployment and has the most vibrant community. The most popular language used to develop Ethereum smart contracts is Solidity with 8,000+ public repositories on GitHub. Solidity is a JavaScript-like, Turing-complete language and has been in constant development since 2014 by the Ethereum team. Solidity gives developers a lot of flexibility, but this flexibility has also resulted in many security flaws. As a result, developers were looking to develop a new language that limits security flaws and came up with Vyper. Vyper is a pythonic programming language that is non-Turing complete, and is focused on security, language and compiler simplicity, and auditability. Hence, Vyper does not have many features that Solidity has like inline Assembly. Vyper is much newer than Solidity, and the first stable version of Vyper was released in 2020. There are currently only 84 public Vyper repositories on GitHub.

This project will explore how difficult it is to learn to develop a smart contract in Vyper, given its smaller community and limited examples on GitHub. This project will also explore the differences between Solidity and Vyper, and the benefits and drawbacks of these differences. To answer these questions, I will reimplement the Uniswap V2 smart contract in Vyper. Uniswap V2 smart contract is a smart contract written in Solidity that facilitates swapping between cryptocurrencies, and it is one of the most widely used smart contracts to date on Ethereum. I chose to reimplement an existing Solidity smart contract because the features are well defined making it easier to test correctness. I also chose Uniswap V2 instead of Uniswap V3 because V3 has additional features for trading. Reimplementing these additional features would involve writing additional application logic, but does not add value to exploring the Vyper. During this project, I will be documenting the amount of time it took to set up the environment, read documentation, develop the code, etc. Additionally, I will document any obstacles I ran into.

1.2 Novel, Unique, Interesting

This project is novel because it explores an up-and-coming language for smart contract development. Vyper is a pythonic programming language, and Python is currently the most popular programming language. This makes it easier for newcomers to learn. Also, Vyper's focus on security, language simplicity, and auditability makes smart contracts more secure. These factors make Vyper a contender to challenge the dominance of Solidity in smart contract development.

1.3 Value and availability to the target user community

The target audience for this project is experienced developers who want to learn smart contract development. Blockchains have seen significant growth in the past few years. This is especially true in 2021, a year when Metamask, a crypto wallet company, saw monthly active users grow from 500,000 to 10,000,000. Given this growth, many developers are going into smart contract development. This project will lay out the differences between Solidity and Vyper to help aspiring smart contract developers choose which language to learn.

This project will be highly available to the target user community because all code will be uploaded to a public GitHub repo. In addition, all the code is based on public resources like Vyper documentation, Solidity documentation, Uniswap V2 public GitHub repo, etc, and the references are included in the README of this project. This will allow others to replicate the project on their own.

2. Research Questions

RQ1: How difficult is it to learn smart contract development with Vyper when there are limited examples on GitHub and answers on Stack Overflow?

The answer to this question varies based on each person's prior knowledge. Hence, before I answer this question, I will provide a quick overview of my background. I worked for a couple of years as a Data Engineer and am about to complete an MSCS from Columbia. I am most familiar with Python, C, and Java. I have also been a user of blockchains since 2021 and have a general understanding of blockchains and smart contracts.

Development Environment

To begin developing in Vyper, a development environment must first be set up, which is often a daunting task for newcomers. Furthermore, a development environment for smart contract development has the added difficulty of having to interact and test with a blockchain. Hence, this made me gravitate towards Remix IDE (Remix), which I previously knew existed. Remix is a web-based IDE that does not require installing additional software and allows writing, compiling, and manually testing smart contracts. Remix is also mentioned in Vyper's documentation. Remix by default works with Solidity, and fortunately, a Vyper plugin can be installed to compile Vyper contracts. With the plugin, I was able to quickly deploy and test code that prints out "hello world".

Unfortunately, Remix quickly felt lacking as I began to reimplement Uniswap V2. For example, compilation errors do not appear on the UI but instead are found in Google Chrome DevTools. Figure 1 shows how compilation errors look like in Google Chrome DevTools. This is because the Vyper plugin sends an HTTP request to a remote compiler to compile the code. The compiled code or errors are found in the response, but the Vyper plugin does not

have the ability to display the results on the UI. This made debugging very tedious. Additionally, the remote compiler does not have the latest Vyper version. This made development confusing because I was getting compilation errors even though I was following the documentation closely. Lastly, Uniswap uses interfaces extensively in its code, but Remix does not support importing interfaces. I originally thought I was incorrectly implementing interfaces, but after multiple tries and countless hours, I realized it was actually an issue with the Vyper plugin. The Vyper plugin is not able to reference other files in the same directory. These issues highlight the lack of tool support for Vyper and made it much more difficult to learn Vyper. As a result, I decided to search for a more stable development environment.

Figure 1: Vyper compilation error in Google Chrome DevTools for Remix

For a more stable development environment, I installed Vyper locally and installed Brownie, a Python-based development and testing framework for smart contracts mentioned in the documentation. I also used a Vyper extension in Visual Studio Code (VS Code) that helps highlight syntax errors and auto-compiles the code for any compilation errors. The only hiccup with this development environment was having compilation differences between Brownie and VS Code. I found that Brownie was using Vyper 0.3.1 and VS Code was using Vyper 0.3.2. I am unsure why Brownie was using version 0.3.1, but it may be because I downloaded Brownie a couple of days before the version 0.3.2 release. All I had to do was add a Brownie config file to the root directory with the desired version of Vyper to resolve this issue. This was clearly documented in Brownie's documentation. Once this environment was set up, Vyper development went smoothly

Overall, I didn't have too much difficulty setting up a development environment for Vyper. Initially trying to develop on Remix made this process more challenging due to the lack of documentation. For example, I couldn't find any Remix documentation on how to find compilation errors and the Vyper plugin's inability to reference other files. However, once I built my local environment, it became much less challenging because Brownie was well documented. Furthermore, I learned that having a consistent Vyper version across development and testing is very important. Vyper is constantly changing, and the difference of 0.0.1 could lead to compilation differences. This is unlike Python where the same code is likely to successfully run across multiple minor versions of Python.

Vyper Development

One of the reasons I chose to reimplement Uniswap V2 instead of writing a brand new contract is because I can focus on the Vyper language and not worry about business logic. During reimplementation, Vyper's pythonic style not only made it feel like I was coding in Python but also made reading the documentation easier. For example, the syntax for defining a function in Vyper is the same as how functions are defined in Python. The semantics between the two languages are similar too. For example, __init__() is called when a new smart contract is deployed. This is similar to how __init__() is called when objects are created in Python. Vyper may even be a simpler language compared to Python because it doesn't have features like inheritance and exceptions. With regards to testing, Brownie uses

pytest, so testing Vyper functions is the same as testing Python functions. People with prior Python experience will find Vyper easy to pick up.

With regards to Vyper's blockchain-related features, I found them intuitive and well-documented. However, this may be an unforeseen benefit of reimplementing Uniswap V2 because it requires me to understand what the Solidity smart contracts were doing before writing it in Vyper. This required me to Google keywords, built-in functions, and concepts I didn't understand. Fortunately, I didn't encounter too many obstacles because there is a large community in Solidity and concepts are well documented. Moreover, Uniswap V2 is the canonical Solidity smart contract and there is a tutorial dedicated to Uniswap V2 on Ethereum's website. After understanding the blockchain-related features in Solidity, all I had to do was find equivalent features in Vyper when reimplementing. However, this doesn't mean that every feature in Solidity exists in Vyper. Vyper has a smaller subset of features because of its newness and its focus on security. For features that exist in both Solidity and Vyper, there are only a handful of features where the languages have different implementations. For example, when a smart contract creates another smart contract in Solidity, the constructor is called by default, but in Vyper's case, Vyper skips the __init__() function. In most cases, Vyper's features are implemented similarly to Solidity's features.

On the other hand, if someone decided to begin smart contract development with Vyper, blockchain-related features may be more difficult to grasp because of fewer documentation and tutorials explaining concepts in Solidity. Newcomers learning smart contract development through Vyper may find it easier to first find the equivalent feature in Solidity and then research the Solidity feature.

Summary

After examining the process of setting up a development environment for Vyper and developing in Vyper, I conclude there is a large enough community and enough documentation to learn smart contract development through Vyper. However, for a better understanding of smart contracts, I would recommend learning Solidity first because there is a much larger community, more documentation and tutorials, and more example code.

RQ2: What are the benefits and drawbacks of differences between Vyper and Solidity?

Vyper's focus on security, language and compiler simplicity, and auditability means it has different features compared to Solidity. Some but not all of these differences are listed in Vyper's documentation. For the features that are listed, the reasonings behind its choice of features are presented, but the drawbacks of these features are not explored. This section will explore the benefits and drawbacks of some of Vyper's features.

Differences listed in Vyper's documentation

No Modifier

Modifiers in Solidity are analogous to decorators in Python, and it allows developers to wrap a function with code before and after it. Vyper decided to remove this feature because a modifier can be defined far away from the functions that use it, making it difficult to audit. I agree with Vyper's design choice. Yet, I think there are use cases for this feature. For example, in Figure 2, Uniswap uses modifiers to define a contract-wide lock to enforce atomicity and prevent re-entry attacks. Re-entry attacks are common security flaws in smart contracts. Recently, the Fei Protocol lost $80 million in assets due to a re-entrancy vulnerability. If modifiers aren't allowed, Vyper should provide a lock feature. This will

prevent mistakes in implementing locks and reduce the amount of duplicate code in the contract.

```
uint private unlocked = 1;
modifier lock() {
    require(unlocked == 1, 'UniswapV2: LOCKED');
    unlocked = 0;
    _;
    unlocked = 1;
}
```

Figure 2: Modifier in Solidity

No Inheritance

Solidity does not support inheritance of smart contracts by other smart contracts. The reasoning for no inheritance follows the reasoning for no modifier. Vyper does not want functions that have interactions to be defined in different files because it decreases auditability. I understand Vyper's design choice, but this could lead to bloated and un-modularized code that makes it difficult for readers to understand. For example, the UniswapV2Pair contract inherits the UniswapV2ERC20 contract. This allows the Uniswap team to separate pair trading logic and the ERC20 logic into two separate files. I would argue this structure made the Uniswap V2 Solidity more understandable. In addition, the situation where two different contracts inherit the same contract would result in large duplication of code.

Support for decimal fixed point numbers

There are only benefits to Vyper's decision to support decimal types. Solidity by default only supports integers. As a result, UniswapV2 had to import custom libraries to support decimals. Custom libraries developed by the smart contract developer or found online could have bugs that lead to vulnerabilities. Additionally, fixed point numbers made reimplementing Uniswap V2 much easier.

No inline assembly

Solidity allows developers to put assembly code into smart contracts. This allows developers to optimize for gas fees when the contract is called. Gas fees are transaction costs paid by the user to the miners for interacting with smart contracts or deploying smart contracts. Vyper removed this feature because assembly code is difficult to read and audit, which can result in vulnerabilities. Figure 3 shows assembly is inserted into Solidity. From the code snippet, it is unclear what the bytecode is doing making it difficult to audit. The benefits of not allowing assembly code in smart contracts far outweigh the gas-saving from using assembly code.

```
assembly {
    pair := create2(0, add(bytecode, 32), mload(bytecode), salt)
}
```

Figure 3: Inline assembly in Solidity

<u>Differences not listed in Vyper's documentation</u>

Using a smart contract to deploy another smart contract

The way new contracts are created from another contract in Solidity is similar to how objects are created in Java, which is with the **new** keyword. In Vyper,

create_forwarder_to([address of master contract]) has to be used to create another contract. Moreover, the master copy of the contract must already exist on the blockchain, and the address of the master copy must be passed into the function. A major drawback is the extra gas fees required to first deploy the master contract. Smart contracts are big objects in the blockchain realm and deploying a smart contract can cost over $10,000.

Operations are restricted for types less than uint256

Vyper is a statically typed language. A developer defines an integer by putting the number of bits used to represent an int. The number of bits has to be a multiple of 8. Vyper only allows operation between objects that are of type uint256. It is unclear why Vyper made this design choice, but it makes operating on variables that are not uint256 cumbersome because of the conversions between types.

3. Methodology

For this project, I reimplemented Uniswap V2 line by line in Vyper. To test the implementation, I wrote tests that covered over 90% of each of the files. Results from test coverage can be found in figure 4, they can also be regenerated with instructions in README. The Token contract has low coverage because test cases were not created for it. The Token contract is a bare-bone ERC20 contract found from a Brownie extension and used only to test the UniswapV2Factory and UniswapV2Pair contracts.

While reimplementation, I kept track of the hours spent on reimplementation. A work log can be found in table 1 in the appendix. I also kept track of the concepts I had to search for which can be found in table 2 in the appendix.

```
tests/test_erc20.py ..                                                   [  8%]
tests/test_factory.py .......                                            [ 39%]
tests/test_pair.py ........                                              [ 73%]
tests/test_pair_feeTo.py ......                                          [100%]
---------- Coverage ----------

  contract: Token - 37.5%
    Token._transfer - 75.0%
    Token.transferFrom - 0.0%

  contract: UniswapV2Factory - 100.0%
    UniswapV2Factory.createPair - 100.0%

  contract: UniswapV2Pair - 90.9%
    UniswapV2Pair._update - 100.0%
    UniswapV2Pair.burn - 100.0%
    UniswapV2Pair.mint - 100.0%
    UniswapV2Pair.swap - 100.0%
    UniswapV2Pair._mintFee - 57.3%
```

Figure 4: Test coverage

4. Datasets

All code and deliverables for the project can be found in the main branch of https://github.com/marv-chan/COMS_6156_Final_Project. README includes instructions on how to install the necessary dependencies and the commands used to run the code. Work log and topics searched can be found in link 1 of the appendix.

5. Reproducibility

This project is highly reproducible because it is based on public sources like Vyper documentation, Solidity Documentation, Uniswap V2 GitHub Repo, Ethereum's website, tutorials, etc. However, the answers to the research questions may vary based on the previous background of the developer.

6. What I learned

Through this project, I was able to familiarize myself with the languages used for smart contract development. Given that Vyper is the main focus of this project, I was able to have a strong grasp of the language. Furthermore, I was also able to indirectly familiarize myself with Solidity through reimplementing Uniswap V2. By studying the languages side-by-side, I am able to understand the potential security vulnerabilities Solidity has through Vyper's design choices.

Uniswap V2 is the canonical example used for smart contract development, and it has been an amazing learning experience to be able to dig into this piece of code. Through reimplementation, I have learned a lot about the mechanics of smart contract development such as the ERC-20 Token Standard, which governs how tokens are implemented under the hood, and how the factory method pattern is utilized in smart contracts. Though Uniswap V2 has in total less than 1,000 lines, it has facilitated hundreds of billions of dollars in trading. It is fascinating how a well-written piece of code in this new programming paradigm can have such an outsized impact. After graduating, I plan to enter the blockchain industry, and I think this project has given me a great start.

7. Appendix

Date	Start	End	Hours
4/11	2:00 PM	7:00 PM	5
4/15	3:00 PM	6:00 PM	3
4/21	2:00 PM	5:00 PM	3
4/22	2:00 PM	6:00 PM	4
4/23	8:00 PM	11:00 PM	3
4/24	10:00 AM	1:00 PM	3
		Total Hours	21

Table 1: Work log

Searches
compiling in Vyper
interfaces
events
memory vs. storage variables
pure vs. view functions
ternary operator
comparing addresses
dynamic arrays
init

create new contract with an existing contract
testing
compiling with different version of Vyper
operations with uint32
application binary interface (ABI)
keccak256
ecrecover

Table 2: Google searches made while reimplementation

8. References

1. Vyper Documentation: https://vyper.readthedocs.io
2. Vyper GitHub Repo: https://github.com/vyperlang/vyper
3. Solidity Documentation: https://docs.soliditylang.org
4. Solidty – Vyper Cheatsheet: https://reference.auditless.com/cheatsheet/
5. Brownie Documentation: https://eth-brownie.readthedocs.io
6. Uniswap V2 Core Contracts: https://github.com/Uniswap/v2-core
7. Ethereum Website: https://ethereum.org
8. Remix IDE: https://remix.ethereum.org
9. Vyper tutorial: https://bowtiedisland.com/vyper-for-beginners-introduction/
10. DAO Hack: https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee
11. Wormhole Hack: https://www.theverge.com/2022/2/3/22916111/wormhole-hack-github-error-325-million-theft-ethereum-solana
12. Fei Protocol Hack: https://www.coindesk.com/business/2022/04/30/defi-lender-rari-capitalfei-loses-80m-in-hack/