CUNY School of Professional Studies

Master of Science in Data Science

Smart Contract Derivatives on the Ethereum Virtual Machine

Samuel Reeves
Candidate

Prof. Nasrin Khansari
Advisor

Abstract

A general purpose blockchain can host smart contracts that can function autonomously. The assets created here calculate a payment amount based on a number of predetermined factors and send it to the previous owner at the time of a trade. There is no other way to get value in or out. When the asset reaches maturity, a failed trade will push all the remaining value to the final owner. Valuing these assets in terms of fiat currency is non-trivial.

Contents

1 Introduction

1.1 What are we building?

For this project, we are testing a smart contract that pays a portion of its initial value to its previous owner whenever it is given to a new one. It should be secure, and it should function for a fixed period of time. It should be generic, meaning that it can be edited or deployed by anyone to a blockchain that uses the Ethereum Virtual Machine. At the time of maturity, any remaining value should be paid back to the final owner, and it should destroy itself.

Payments should be weighted by a predefined function, but should never allow the contract to go bankrupt. Because of these unique features, there is no risk at all to the value in the contract, and there is no need to rely on an exchange or intermediary. One person should be able to deploy this derivative and sell it without any 3rd party involved at any point in its life cycle. As long as the blockchain survives, the derivative should continue to function autonomously, leaving the possibility for all kinds of custom payment structures or perhaps a time-to-maturity of a thousand years.

1.2 Why the Ethereum Virtual Machine?

The Ethereum Virtual Machine (EVM) is a robust general purpose system for deploying smart contracts in a block chain ledger. Smart contracts on the EVM don't run in real time, but they can respond to messages sent across the network from valid addresses. They can also store small amounts of information. The EVM offers us some basic data types and algebraic operations, and the ability to generate other assets or tokens. [2] It can send quantities of currency autonomously, when prompted to do so.

We are limited to 10 decimal places of floating point math, and we are unable to use trigonometric or calculus functions. Both computing power on the EVM and storage on the network are costly. In order to use this medium effectively, it is necessary to shrink the size of the binary to be deployed and for it to run very efficiently.

So, for our stated purpose, which is to make a transferable smart contract that pays its old owner for the time that they held the contract, it makes a lot of sense to use a state machine on the EVM. However,

there are some serious caveats that will require testing.

2 Asset Payment Structure

A fixed value V_0 is sent to the smart contract during deployment at starting time t_0. The timestamp at the end of the period when the bond matures, we will call t_m. The time t_n is the timestamp at the time of the n^{th} transaction.

If a change of owner is initiated at $t_n > t_m$, a trade is voided, and any remaining value returns to the wallet of the owner of the bond. Then, the bond is burned; it removes itself from the blockchain and is gone forever.

The generalized structure is this:

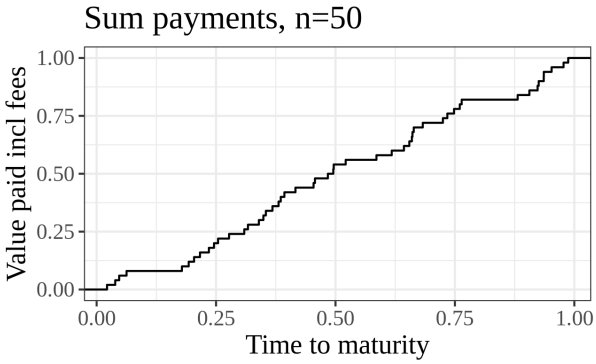
$$t_\mu = \text{floor}(\frac{t_m - t_0}{2})$$

$$V_m = V_0 - (\sum_{n=1}^{m} w_n V_0 - g_n - \varepsilon_n)$$

Where, $W_n V_0$ is the weighted payment that arrives at the seller's wallet before gas fee g_n, and minus any small error in the floating point math. V_m is the amount left over to the final owner of the bond, which in simulation has been consistently about 5% of V_0. The weighting coefficient w_n gives the bond the shape of the curve of its payout structure. Altering the mechanism for weighting the payouts will necessarily influence the amount left at t_m.

2.1 Uniformly distributed payouts

For a uniformly distributed payout structure, w_n is just the percentage of time total time relative to maturity that the bond was owned by the seller.

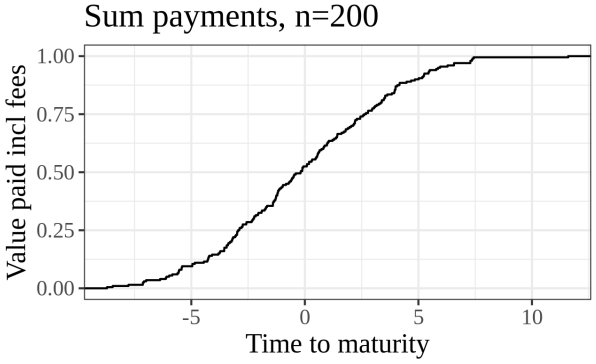


$$w_n = \frac{t_n - t_{n-1}}{t_m - t_0}$$

There is no curve, and assuming the transaction costs G are negligible, the magnitude of payouts is just a flat line from t_0 to t_M.

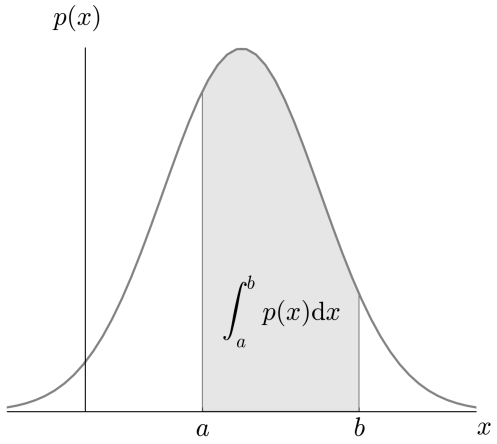
2.2 Normally distributed payouts

Another way to define a fixed payment structure could be to weight the payments based on a curve. In the discrete uniform case, the weight of the payout is a function of the area under a uniform normal curve for the time period the bond has been held relative to the total time $t_0 \to t_m$.



The time period is weighted continuously from beginning to end by taking the P-value of the Z score for the time period. If you buy the asset close to t_0 and you sell it soon after, the payout will be very small,

but the sale price for the asset will be considerably large because the majority of the principle is intact. If you own it and sell it closed to t_μ, the payouts will be larger for the same amount of time, and the sale price will be considerably smaller with only the right tail of the distribution left to pay out.



There are two cases. The weighting function is defined piecewise for areas where t_n and t_{n-1} fall on either side of t_{mu}, the second part of the function describes an area that falls completely on one side of t_{mu}, inclusive. These functions are dependent on the practical method of approximating the Cumulative Distribution Function.

$$w_n = \begin{cases} 1 - P(z_{n-1}) - P(z_n) & t_{n-1} < t_\mu < t_n \\ \left| P(z_n) - P(z_{n-1}) \right| & \text{otherwise} \end{cases}$$

We arrive at a Z-score in a familiar way [3],

$$\sigma = \frac{t_M - t_0}{\sqrt{12}}$$

$$z_n = \left| \frac{t - t_\mu}{\sigma} \right|$$

Because of the limitations of the EVM, it is necessary to use a numerical approximation for the P value which will determine the actual amount of payment x_t. We are unable to use trigonometric or calculus functions. Any chosen approximation with these constraints will be a

compromise among low error rate across the spectrum, easily definable constants, and ease of computation.

This presents a theoretical problem, but not a practical one. Time flows in one direction only, and these values for the transaction amounts are calculated at the time the transaction is initiated.

2.3 Approximating a closed-form

For some problems, selecting a method for algebraic approximation may be a matter of taste. As stated, we are bound by the necessity to optimize both computational simplicity and size in bytes of the smart contract binary. It's also crucial that the asset never at any point before t_m pays out all of its value or goes into negative value territory.

We can use a simple algorithm such as this algorithm published by Polya in 1945, based on the initial forms from LaPlace and Euler [6],

$$P(z) \approx \frac{1}{2}(1 + \sqrt{1 - e^{\frac{-2z^2}{\pi}}})$$

or this "pocket calculator" version from Lin in 1990 [5],

$$y = 4.2\pi\frac{z}{9-z}, \qquad 0 \leq z \leq 9$$

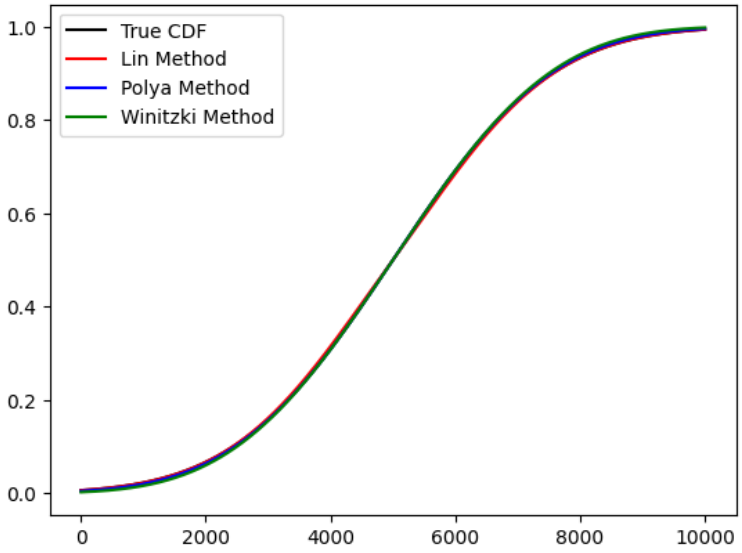
$$P(z) \approx 1 - \frac{1}{1+e^y}$$

or a later one such as this one from Winitzki in 2008 [7]:

$$P(z) \approx \frac{1}{2}\left[1 + \left(1 - \exp\frac{-(\frac{z^2}{2})(\frac{4}{\pi} + 0.147\frac{z^2}{2}}{1 + 0.147\frac{z^2}{2}}\right)^{\frac{1}{2}}\right]$$

It's not immediately clear which is the best compromise, and there are over 20 published variations if we include the ones that make use of trigonometric terms. [8]

After testing these three in simulation, it appears that only the Lin method is suitable. It has the most predictable error rate, consistently producing a total area under the curve near 94% of the true area. Although the other two sometimes exceed this accuracy to the level of 98%, they also both have an unpredictable tendency to estimate around 30% of the true area.



Our simulation consisted of making an asset with a value of 1 unit and a time to maturity of 10,000 discrete time values. Trades were conducted at random moments during this interval until the time of maturity, and the total amount of the simulated payouts was compared with the known true value of the area under the curve. This was performed a hundred times for each approximation method, with the average total number of trades performed between 5 and 20.

The Lin method is more consistent and quicker to calculate using our limited computing resources. As a way of simplifying this process, we can include constants such as $4.2\pi = 13.1946891451$, conscious of the 10-place precision of our floating point math allowed in Vyper. Floating point precision errors or edge cases involving trades performed in very small discrete time intervals likely account for the inconsistent results with the Polya and Winitzki methods. It is known that each approximation is idiosyncratic in its absolute error function; the error rates have greatly fluctuating orders of magnitude depending on the region of the curve. [8]

3 The Smart Contract

3.1 High-level languages and compiling for EVM

In order to engineer a smart contract, one must write a contract which describes all relevant logic and nothing else. Generally, this part is done in a high-level language, though lower level options (Huff and Yul are closer to the bytecodes that would run in EVM, but they do not provide many failsafes for developers). Then, the contract is compiled down to a specific type of binary known as Application Binary Interface (ABI). The binary file can be pushed to the network via an established full node, and it will remain there with an address until it is permanently destroyed. [2] Vyper and Solidity are the only serious choices in higher-level programming at the current time, and here we select Vyper for a few of its aesthetic and practical qualities.

Vyper is a newer language compared with the standard Solidity, made available with many of the "creature comforts" of Python. By design, it is considerably less capable. It does not feature infinite lists at the time of this paper, and many other features available in Solidity or Python are not granted in Vyper for security reasons. [1] They say that Vyper is better at computing an upper bound for gas fees, but I do not know if this is true. It is also rumored to compile down to binaries that are up to 40% smaller than equivalent ones compiled from Solidity. [4] As a simpler language with a smaller "instruction set", this stands to reason.

The generic asset described here only has one externally callable method: give. At the time of deployment, the contract is loaded with value. After this point, which we call t_0, the owner can call only the give function to register a new owner to the smart contract. The only way to receive a payout from the contract is to give ownership to somebody else. If this asset is held after the date of maturity, t_m, the give function will not register a new owner. Instead, it will send all remaining value to the owner and permanently delete itself.

3.2 Gas Fees

Since Ethereum is a decentralized network made of nodes that are doing work to establish consensus, any smart contract will live on other people's machines and use other people's electricity.

There is an initial fee to pay for deployment of the contract, which physically stores it somewhere on the network. Since this will happen before it is loaded with value and becomes transferable, it is not included in the simulation. The smart contract is not "aware" of this fee, which must come from a previously established address. Still, it is relevant to the overall valuation because it is the largest significant loss of value in the life cycle of the asset. Gas fees will also be paid in the process of delivering payouts when the give function is called.

With a finite number of trades and a reasonably large initial value, gas fees are negligible:

$$\lim_{V_0 \to \infty} \frac{G}{V_0} = 0$$

4 Conclusion

By using smart contracts in this way, an individual actor can design, deploy, and trade assets without the need for any form of centralized exchange. In this way, they differ from the Non-Fungible Token ecosystem. By using a base cryptocurrency we give the smart contract a value that it can disperse completely autonomously when prompted to do so by its owner. This leaves the process of valuation and sale completely external. It is possible to list these things on an exchange and further automate the process, but it may be more beneficial to buy and sell such devices without using the internet at all.

Valuation in the use case we have shown is not trivial. The valuation today in fiat currency of one of these derivatives is not necessarily equal to the remaining value in cryptocurrency adjusted for the exchange rate today. The period of maturation, the fluctuations in the exchange rate, fluctuations in transaction costs after the time of deployment, and the error rate from the approximation of normally distributed payments (this remainder will belong to the final owner), and the discounted time-value of money will all play in some significant way to give us a calculable value. If the time to maturity is far in the future, a historical average for the crypto-fiat exchange rate will probably not be very useful. Using the exchange rate close to the time of deployment will also not be very useful because the magnitude of payouts is smallest at the ends of the two tails. This indeterminacy is potentially very good for making a market; people can very easily disagree on a valuation.

However, it may present some issues for finding a buyer at the very beginning of the asset's life cycle.

There are serious limits to the capabilities of decentralized apps using the EVM. Perhaps this will change in the future. At the moment, the process of engineering things for this medium is very similar to engineering using microcontrollers. In fact, the limitations on binary size and the skeletal instruction set make it feel a lot like programming an ATtiny85 chip with its 8KB of flash memory, 512B of EEPROM, limited clock speed, and limited I/O capabilities. Compiling Arduino code down to a binary, flashing it onto the chip, and then testing manually is almost exactly like the decentralized app workflow, with the only major difference that there are more established C/Arduino libraries available for import.

An important consideration is the potential for smart contracts to interact with one another. For example, if these devices are produced in large amounts, it isn't logical to always include the helper functions in the code's preamble. Deployment and data storage are the greatest costs in the process, so it makes much more sense to deploy, for example, one smart contract that calculates μ and σ, another that calculates a Z-score, and a third that only performs the Lin approximation based on two of these values. Only one copy of each tool is needed on the decentralized network to service an arbitrary number of bonds. They can be coerced to perform and return calculations directly to each other in a chain for a gas fee that is considerably smaller than the added cost of deployment. Determining the validity of this strategy depending on transaction costs and the length of each binary is a potentially useful new area of research.

5 Code Appendix

5.1 Simulation

```
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt
import pandas as pd

np.random.seed(hash(float('inf')))
```

```
_e = 2.7182818284

def z_score(t, mu, sd):
    return abs((t - mu) / sd)

def Polya_tail(z):
    y = 1 - _e ** (-0.63661977236 * z ** 2)
    return (1 + np.sqrt(y)) / 2

def Winitzki_tail(z):
    a = -z ** 2 / 2
    b = 0.147 * a
    c = 1 - _e ** (a * (1.27323954474 + \
        b) / (1 + b))
    return 0.5 * (1 + c ** 0.5)

def Lin_tail(z):
    y = 13.1946891451 * z / (9 - z)
    return 1 - 1 / (1 + _e ** y)

def calculate_w(t, zp):
    zn = z_score(t)
    w = zp - zn
    return w

print(z_score(4, 6, 1)) # 2 -- 98%
Lin_tail(2), Polya_tail(2), Winitzki_tail(2)

table = pd.DataFrame(
         columns=['t', 'z', 'cdf',
                  'lin', 'polya', 'winitzki'])

# Integer time, Z score, and true cumulative
# distribution function
table['t'] = range(0, 10000)
table['z'] = z_score(table['t'], 5000, 2000)
table['cdf'] = norm.cdf(table['z'])
```

```
# 3 numerical methods for approximation of the CDF
table['lin'] = Lin_tail(table['z'])
table['polya'] = Polya_tail(table['z'])
table['winitzki'] = Winitzki_tail(table['z'])

table.iloc[:5000, 2:] = 1 - table.iloc[0:5000, 2:]
table.head()

plt.plot(table['t'], table['cdf'],
         color='black', label = 'True CDF')
plt.plot(table['t'], table['lin'],
         color='red', label = 'Lin Method')
plt.plot(table['t'], table['polya'],
         color='blue', label = 'Polya Method')
plt.plot(table['t'], table['winitzki'],
         color='green', label = 'Winitzki Method')
plt.legend()

# Create n random time intervals with Z score
def make_slices(n=20, T=10000, mu=5000, sd=2000):

    # Start on the left side of the curve
    t = [0]
    z = [-2.50]

    # Pick random times for trades within the T
    for i in range(n):
        t.append(np.random.randint(t[-1]+1, T))
        if t[-1] >= T-1:
            break
        elif t[-1] < mu:
            z.append(0 - z_score(t[-1], mu, sd))
        else:
            z.append(z_score(t[-1], mu, sd))
    return t,z

def make_test(n=100):
```

```
    t,z = make_slices(n)
    trades = []
    for i in range(len(z) - 1):
        left = Lin_tail(z[i])
        right = Lin_tail(z[i+1])
        trades.append(abs(left - right))
    return trades

sum(make_test())
```

5.2 Vyper contract

5.3 Binary file

References

[1] URL: https://vyper.readthedocs.io/en/v0.3.7/.

[2] @wackerow. *Anatomy of Smart Contracts*. Aug. 2022. URL: https://ethereum.org/en/developers/docs/smart-contracts/anatomy/.

[3] Hervé Abdi. "Z-scores". In: *Encyclopedia of measurement and statistics* 3 (2007), pp. 1055–1058.

[4] Patrick Collins. *Patrickalphac/SC-language-comparison*. Sept. 2022. URL: https://github.com/PatrickAlphaC/sc-language-comparison.

[5] Jinn-Tyan Lin. "Pocket-calculator approximation to the normal tail". In: *Probability in the Engineering and Informational Sciences* 4.4 (1990), pp. 531–533.

[6] George Pólya et al. "Remarks on computing the probability integral in one and two dimensions". In: *Proceedings of the 1st Berkeley symposium on mathematical statistics and probability*. 1945, pp. 63–78.

[7] Sergei Winitzki. "A handy approximation for the error function and its inverse". In: *A lecture note obtained through private communication* (2008).

[8] Ramu Yerukala and Naveen Kumar Boiroju. "Approximations to standard normal distribution function". In: *International Journal of Scientific & Engineering Research* 6.4 (2015), pp. 515–518.