INTRODUCING FRAMECHAINS 1

Granular Anomaly Detection

Columbia University, 2024

Abstract

The world of Blockchain has been plagued by one primary problem since it's inception - The Double Spending problem. Double spending means a person can use a coin, and since blockchain is a decentralized peer to peer network, maintain a copy of the coin to utilize again, hence, spending the coin twice or more. The first solution was Proof of Work, which was rather resource intensive, and hence provided rewards for the ones mining through resources.

Then, arrived Proof of Stake, which defies the philosophy of Blockchain as a balancing financial system, by bestowing the power to validite only to the one with resources to be staked. The notion of decentralization being the birth of a new world where governance and power are not contained in a central powerful entity, Proof of Stake reversed the progress to the old philosophy of power centralized to the king/-class with the resources.

In paper 0, I introduced Frames as a datastructure to sustain a naturally linking ledger. I follow up with this paper to introduce Granular Anomaly Detection (GAD), as the main consensus mechanism (solution to the Double Spending problem), in the implementation of FrameChains. Granular Anomaly Detection implies the breaking down of the validation system in three small units and detecting any anomalies in the smallest transaction process possible. Then, a Zero Knowledge Proof system is implemented to ascertain the truth value of the transaction, thus, enabling transactions that are localized to the transitors.

Local Ledger

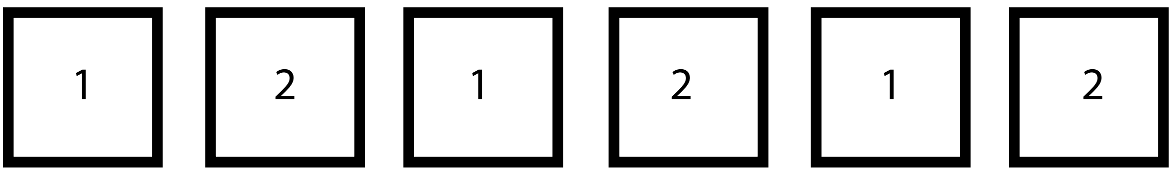
Granular Anomaly Detection begins with a local ledger implemented on each node to record all transactions that pass through that specific node.

The datastructure for this implementation is a MirrorStack. Here is a sample written in Python.

```
## Written by -YON-
## Based on Implementation of Stack which modified Interface from Dr. Brian Borowski's Data Structures Class of '24 at Columbia University

class MirrorStack:
    def __init__(self):
        self.size = 0
        self.data = []

    def isEmpty(self):
        if self.size == 0:
            return True
        else:
            return False

    def get_size(self):
        return self.size

    def push(self, element_to_be_pushed):
        self.size = self.size + 1
        self.data.append(element_to_be_pushed)
```

```
def pop(self):
        if self.size == 0:
            raise Exception("The Stack is Empty. Nothing to Pop.")

        last_index = self.size - 1
        return self.data.pop(last_index)

    def peek(self):
        if self.size == 0:
            raise Exception("The Stack is Empty. Nothing to Peek.")

        last_index = self.size - 1
        return self.data[last_index]

    def mirror(self, other_stack):
        if self.size == 0:
            raise Exception("The Base Stack is Empty.")
        if other_stack.get_size() == 0:
            raise Exception("The Incoming Stack is Empty.")
        greater_size = 0
        if self.size >= other_stack.get_size():
            greater_size = self.size
        else:
            greater_size = other_stack.get_size()
        response_stack = MirrorStack()
        for i in range(greater_size):
            if i < self.size:
                response_stack.push(self.data[i])
            if i < other_stack.get_size():
                response_stack.push(other_stack.data[i])
        return response_stack

    def validate(self):
        for i in range(len(self.data)):

            if i == 0:
                continue

            if i == len(self.data) - 1:
                continue

            print(self.data[i]["Validation_Function"](self.data[i-1], self.data[i+1]))
```

The first key feature here is the Mirror function which takes another of the Local ledgers and puts one node after the other as though shuffling a deck of cards.



The Mirror function responds with the first stack (1) shuffled into the second stack (2). Now each node has a validate function in them shared in the below code. So when Validate is called, the nodes validate each other signifying the first of the three Granular components.

Unit 1 - Node Validation

While this system still requires perfection, the notion is each node contains it's own validation function that validates the flow from the previous node to the next node stored in the ledger. This is to establish a layer of trust between the two transactees that are undergoing this process.

The following code is a small sample implementation of the node validation system. At the end, True will be printed by each node as it validates the flow of the one that came before it and the one that came after.

```
from hashlib import sha256
from uuid import uuid1, uuid4
from MirrorStack import MirrorStack

def sha_256(input):
    return sha256(input.encode('utf-8')).hexdigest()

def generate_public_key():
    return sha_256(str(uuid1()))

def generate_private_key():
    return sha_256(str(uuid4()))

User_One = {
    "Public_Key": generate_public_key(),
    "Private_Key": generate_private_key(),
    "Crypto_Amount": 5,
    "Personal_Ledger": MirrorStack()
}

User_Two = {
    "Public_Key": generate_public_key(),
    "Private_Key": generate_private_key(),
    "Crypto_Amount": 5,
    "Personal_Ledger": MirrorStack()
}

User_Three = {
    "Public_Key": generate_public_key(),
    "Private_Key": generate_private_key(),
    "Crypto_Amount": 5,
    "Personal_Ledger": MirrorStack()
}

User_Four = {
    "Public_Key": generate_public_key(),
    "Private_Key": generate_private_key(),
    "Crypto_Amount": 5,
    "Personal_Ledger": MirrorStack()
}
```

```
def validated_transaction(sender, receiver, amount):
    if(sender["Crypto_Amount"]<amount):
            raise Exception("Insufficient Funds.")


## Define Validation function
    def validate_link(previous, next):
            if previous["Initial_Amount"] + previous["Transaction_Amount"] == next["Initial_Amount"]:
                return True
            else:
                return False

## Register Sender Frame
    sender["Personal_Ledger"].push(
        {
        "Initial_Amount": sender["Crypto_Amount"],
        "Transaction_Amount": -1 * amount,
        "Validation_Function" : validate_link
        }
    )

## Register Receiver Frame
    receiver["Personal_Ledger"].push(
        {
        "Initial_Amount": receiver["Crypto_Amount"],
        "Transaction_Amount": amount,
        "Validation_Function": validate_link
        }
    )

## Complete Transaction
    sender["Crypto_Amount"] = sender["Crypto_Amount"] - amount
    receiver["Crypto_Amount"] = receiver["Crypto_Amount"] + amount


## Let's Start with Validated Transaction
validated_transaction(User_One, User_Two, 3)
validated_transaction(User_One, User_Two, 2)
validated_transaction(User_Two, User_One, 1)

print(User_One["Personal_Ledger"].data)
print(User_Two["Personal_Ledger"].data)
print(User_One["Personal_Ledger"].mirror(User_Two["Personal_Ledger"]).validate())


print('Block Info')
print('Sender Public Key: ')

print('Receiver Public Key: ')
print('')
```

Unit 2 - Framechain Ledger

A FrameChain ledger will be implemented throughout the network to record transactions, and that will enable a branching ledger that stores values per user. To account for a too large size in the main ledger, nothing will be stored other than a sequence of integers that are the amount of transactions that passed through.

Below is sample python code for a Framechain.

```
class Frame:

    def __init__(self, outer_layer = None, inner_layer = None, creative_mode=True):
        self._external_layer = outer_layer
        self._inner_layer = inner_layer
        self._activated = False
        self._layers = 1
        self._creative_mode = creative_mode
        self._value = None

    ### Creative Mode Operations
    def get_creative_mode(self):
        return self._creative_mode

    def activate_creative_mode(self):
        if self._creative_mode:
            raise Exception("Creative mode already active.")

        self._creative_mode = True
        return

    def deactivate_creative_mode(self):
        if not self._creative_mode:
            raise Exception("Creative mode turned off.")
        self._creative_mode = False
        return

    ### Layer operations
    def get_layers(self):
        return self._layers

        #Increment Layers by One
    def increment_layers(self):
        self._layers += 1

        #Decrement Layers by One
    def decrement_layers(self):
        self._layers -= 1

        #Add to Layer a certain value |
        #Input a negative value in the parameter to subtract
    def sum_to_number_of_layers(self, number_to_sum):
        self._layers += number_to_sum
```

```
#Activates a Frame with a certain number of Inner Frames

def get_activated(self):
    return self._activated

#Inner Frames are 1 if by default
def activate(self, _number_of_inner_frames = 1):
    #If the frame is already activated, Raise an exception
    if self._activated:
        raise Exception("Activating an already active Frame.")

    #Initialize an array of inner layers

    self._inner_layer = []

    #Loop through the number of inner frames required
    for i in range(_number_of_inner_frames):
        #Initialize a new frame with outer layer equaling the current frame
        #Add to the inner layer array
        new_frame = Frame(outer_layer=self)
        self._inner_layer.append(new_frame)

    #set activated to true
    self._activated = True
    #Increment layers
    self._layers += 1

    #Initialize Current Frame to loop through external frames
    current_frame = self

    #Until the last external frame is reached
    while current_frame._external_layer != None:
            #Go to the external frame and increment layers by one
            current_frame = current_frame._external_layer
            current_frame.increment_layers()

    #Return the number of layers from this frame - 2
    return self._layers

def deactivate(self):
    #If the Frame is not activated, raise an exception.
    if not self._activated:
        raise Exception("Deactivating an already inactive Frame.")

    #Subtract one from layers to exclude the frame itself
    layers_inside_this_frame = self._layers - 1

    #Initialize a Frame to loop through external frames and subtract the number of layers inside this frame.
    current_frame = self
    while current_frame._external_layer != None:
            current_frame = current_frame._external_layer
            current_frame.sum_to_number_of_layers((-1 * layers_inside_this_frame))
```

```
        #Reset this frame's values
        self._layers = 1
        self._activated = False
        self._inner_layer = None

        #Return the number of layers - 1
        return self._layers


    #Enters the frame at the specified index
    def enter_frame(self, frame_number_to_enter = 0):

        #If the mode is creative mode
        if self._creative_mode:
            if not self._activated:
                #frame number to enter is an index and starts from 0, so add 1 to it
                self.activate(frame_number_to_enter+1)
        else:
            #If the Frame is not activated, raise an exception.
            if not self._activated:
                raise Exception("Entering an inactive frame. Turn on creative mode if you would like it to be done automatically.")

        #self becomes the inner layer with the frame number to enter
        return self._inner_layer[frame_number_to_enter]

    #Exits frame,
    #number_of_frames inside the external frame if it is to be created by creative mode.
    #current_frame_position inside the bunch of external frames if it is to be created by creative mode.
    def exit_frame(self, number_of_frames = 1, current_frame_position = 0):

        #If the mode is creative mode
        if self._creative_mode:
            if self._external_layer == None:
                if number_of_frames <= current_frame_position:
                    raise Exception("Number of Frames less than Frame Position. Note: current_frame_position is an index, so starts from 0.")

                #Initalize an external frame
                self._external_layer = Frame()

                #Store in a variable
                external_layer = self._external_layer

                #Initialize layers
                external_layer._inner_layer = []
                external_layer._activated = True
                #Create a number_of_frame number of inner frames for the external layer
                for i in range(number_of_frames):
                    new_frame = Frame(outer_layer=external_layer)
                    external_layer._inner_layer.append(new_frame)

                #update layers
                external_layer._layers = self._layers + 1

                #At the index of the current frame position, store the current frame
                external_layer._inner_layer[current_frame_position] = self
        else:
```

```
#If the Frame is not activated, raise an exception.
        if self._external_layer==None:
            raise Exception("No external frame. Turn on creative mode if you would like it to be done automatically.")

    #returns the current frame
    return self._external_layer
```

Unit 3 - Zero Knowledge Sequence Validation

Deriving the two accounts from the ledgers, each node does the mirroring function to generate a sequence of numbers.

Then, a middle linking function will take the hash of the sequence from the Local Ledgers, and the sequence from the mirroring function, and equate them.

If they are equal, then the validation will be granted and the transactions registered. The middle linking function will not know what the sequences will be, it will only get the hashed values from both ends and validate accordingly.

If they don't equate, then, there is a possibility that the two transactees conspired together to validate their local ledgers, and hence the transaction will be declined by the network.

To prevent the centralization of the middle linking function, it will run throughout the network.

Conclusion

What is described primarily through code and statements here is only the basic functionality of GAD. As a security paradigm of breaking down each process to closely detect any anomaly, it can be fortified even more with the creative minds of the Decentralized Finance community.

The vision is that the GAD validation system with Framechains can enable a peer to peer financial network that is capable of processing secure transactions without the need for intensive computation resource, as in the case of proof of work, or a stacking mechanism that while decentralized in it's node implementation, is centralized in the sense that it only empowers the one with resources.

Yon.