THE HIVE

TECHNICAL DOCUMENT

Version 1.0

PAYMON

Paymon - Blockchain Platform

Gleim Semyon

2018

TABLE OF CONTENTS

ABSTRACT	3
INTRODUCTION AND DESCRIPTION OF THE SYSTEM	4
WEIGHTS AND MORE	7
STABILITY OF THE SYSTEM, AND CUTSETS	8
POOLS	12
PROOF-OF-ME	13
EXCHANGE	15
BUSINESS PLATFORM PROFIT	18

ABSTRACT

In this paper we analyze the mathematical foundations of a cryptocurrency PaymonCoin (PMNC). The main feature of this novel cryptocurrency is the hive, a directed acyclic graph (DAG) for storing transactions. The hive naturally succeeds the blockchain as its next evolutionary step, and offers features that are required to establish a machine-to-machine micropayment system. An essential contribution of this paper is a family of Markov Chain Monte Carlo (MCMC) algorithms. These algorithms select attachment sites on the hive for a transaction that has just arrived.

Contact: s.a.gleim@paymon.org

INTRODUCTION AND DESCRIPTION OF THE SYSTEM

Internet selling usually using financial institutions as trustee for electronic payments processing. This system works well enough for most operations, but suffers from the weakness inherent in models based on trust. Bitcoin was supposed to be a system based on cryptographic evidence instead of trust, but in turn, has a number of other disadvantages: the high cost of transactions, a large delay in confirmation, and dependence on miners. With a decrease of miners, the number of confirmed transactions per unit of time will decrease. There is a need in electronic payment system that based on cryptographic evidence, but not dependent of miners.

In this paper we discuss an innovative approach that does not incorporate blockchain technology. This approach is currently being implemented as a cryptocurrency called PaymonCoin. The purpose of this paper is to focus on general features of the hive, and to discuss problems that arise when one attempts to get rid of the blockchain and maintain a distributed ledger. The concrete implementation of the hive protocol is not discussed.

In general, a hive-based cryptocurrency works in the following way. Instead of the global blockchain, there is a DAG that we call the hive. The transactions issued by nodes constitute the site set of the hive graph, which is the ledger for storing transactions.

The edge set of the hive is obtained in the following way: when a new transaction arrives, it must approve or try to approve (we will discuss below) two previous transactions. these approvals are represented by directed edges, as shown in Figure 1. If there is not a directed edge between transaction A and transaction B, but there is a directed path of length at least two from A to B, we say that A indirectly approves B. There is also the “genesis” transaction, which is approved either directly or indirectly by all other transactions. The genesis is described in the following way. In the beginning of the hive, there was an address with a balance that contained all of the tokens. The genesis transaction sent these tokens to several other “founder” addresses. Let us stress that all of the tokens were created in the genesis transaction.

No tokens will be created in the future, and there will be no mining in the sense that miners receive monetary rewards “out of thin air”.

A quick note on terminology: sites are transactions represented on the hive graph. The network is composed of nodes; that is, nodes are entities that issue and validate transactions.

The main idea of the hive is the following: to issue a transaction, users must work to approve other transactions. Therefore, users who issue a transaction are contributing to the network’s security. It is assumed that the nodes check if the approved transactions are not conflicting. If a node finds that a transaction is in conflict with the hive history, the node will not approve the conflicting transaction in either a direct or indirect manner.

As a transaction receives additional approvals, it is accepted by the system with a higher level of confidence. In other words, it will be difficult to make the system accept a double-spending transaction.

It is important to observe that we do not impose any rules for choosing which transactions a node will approve. In order to issue a transaction, a node does the following:

- The node chooses two other transactions to approve according to an algorithm. In general, these two transactions may coincide.
- The node checks if the two transactions are not conflicting, and does not approve conflicting transactions.
- For a node to issue a valid transaction, the node must solve a cryptographic puzzle similar to those in the Bitcoin blockchain. This is achieved by finding a nonce such that the hash of that nonce concatenated with some data from the approved transaction has a particular form. In the case of the Bitcoin protocol, the hash must have at least a predefined number of leading zeros.

It is important to observe that the PMNC network is asynchronous. In general, nodes do not necessarily see the same set of transactions. It should also be noted that the hive may contain conflicting transactions. The nodes do not have to achieve consensus on which valid transactions have the right to be in the ledger, meaning all of them can be in the hive. However, in the case where there are conflicting transactions, the nodes need to decide which transactions will become orphaned.

The main rule that the nodes use for deciding between two conflicting transactions is the following: a node runs the tip selection algorithm many times, and sees which of the two transactions is more likely to be indirectly approved by the selected tip.

Let us also comment on the following question: what motivates the nodes to propagate transactions? Every node calculates some statistics, one of which is how many new transactions are received from a neighbor. If one particular node is “too lazy”, it will be dropped by its neighbors. Therefore, even if a node does not issue transactions, and hence has no direct incentive to share new transactions that approve its own transaction, it still has incentive to participate.

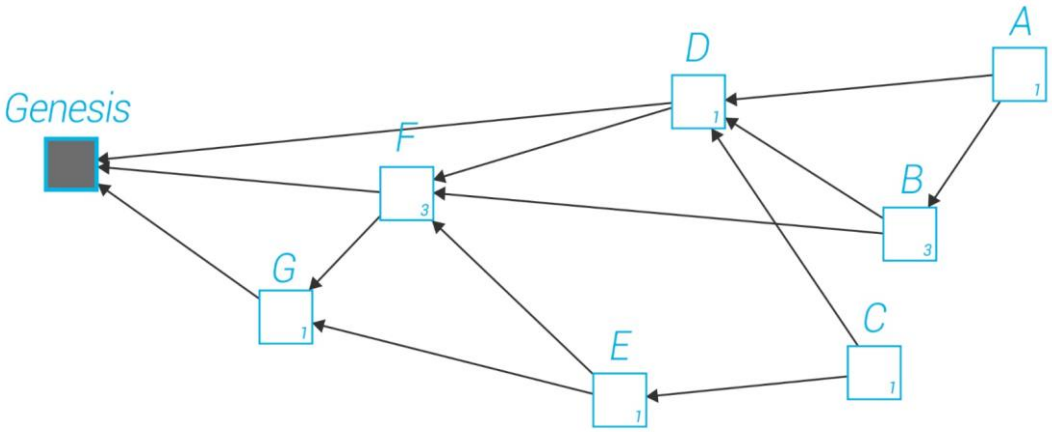

Figure 1 - DAG

WEIGHTS AND MORE

In this section we define the weight of a transaction, and related concepts. The weight of a transaction is proportional to the amount of work that the issuing node invested into it. In the current implementation of PMNC, the weight may only assume values 3n, where n is a positive integer that belongs to some nonempty interval of acceptable values. In fact, it is irrelevant to know how the weight was obtained in practice. In general, the idea is that a transaction with a larger weight is more “important” than a transaction with a smaller weight.

One of the notions we need is the cumulative weight of a transaction: it is defined as the own weight of a particular transaction plus the sum of own weights of all transactions that directly or indirectly approve this transaction.

Let us define “tips” as unapproved transactions in the hive graph.

We need to introduce two additional variables for the discussion of approval algorithms. First, for a transaction site on the hive, we introduce its height - the length of the longest oriented path to the genesis and depth - the length of the longest reverse-oriented path to some tip.

Also, let us introduce the notion of the score. By definition, the score of a transaction is the sum of own weights of all transactions approved by this transaction plus the own weight of the transaction itself.

In order to understand the arguments presented in this paper, one may safely assume that all transactions have an own weight equal to 1. From now on, we stick to this assumption.

STABILITY OF THE SYSTEM, AND CUTSETS

Let $L(t)$ be the total number of tips in the system at time t. One expects that the stochastic process $L(t)$ remains stable. More precisely, one expects the process to be positive recurrent. In particular, positive recurrence implies that the limit of $P[L(t) = k]$ as $t \to \infty$ should exist and be positive for all k ≥ 1. Intuitively, we expect that L(t) should fluctuate around a constant value, and not escape to infinity. If $L(t)$ were to escape to infinity, many unapproved transactions would be left behind.

To analyze the stability properties of $L(t)$, we need to make some assumptions. One assumption is that transactions are issued by a large number of roughly independent entities, so the process of incoming transactions can be modeled by a Poisson point process. Let λ be the rate of that Poisson process. For simplicity, let us assume that this rate remains constant in time. Assume that all devices have approximately the same computing power, and let h be the average time a device needs to perform calculations that are required to issue a transaction. Then, let us assume that all nodes behave in the following way: to issue a transaction, a node chooses two tips at random and approves them. It should be observed that, in general, it is not a good idea for the "honest nodes" to adopt this strategy because it has a number of practical disadvantages. In particular, it does not offer enough protection against "lazy" or malicious nodes. On the other hand, we still consider this model since it is simple to analyze, and may provide insight into the system's behavior for more complicated tip selection strategies.

Next, we make a further simplifying assumption that any node, at the moment when it issues a transaction, observes not the actual state of the hive, but the one exactly h time units ago. This means, in particular, that a transaction attached to the hive at time t only becomes visible to the network at time t+h. We also assume that the number of tips remains roughly stationary in time, and is concentrated around a number $L_0 > 0$. In the following, we will calculate L_0 as a function of λ and h. Observe that, at a given time t we have roughly λh "hidden tips" (which were attached in the time interval $[t - h; t)$ and so are not yet visible to the network); also, assume that typically there are r "revealed tips" (which were attached before time $t - h$), so $L_0 = r + \lambda h$.

By stationarity, we may then assume that at time t there are also around λh sites that were tips at time t − h, but are not tips anymore. Now, think about a new transaction that comes at this moment; then, a transaction it chooses to approve is a tip with probability $\frac{r}{r+\lambda h}$ (since there are around r tips known to the node that issued the transaction, and there are also around λh transactions which are not tips anymore, although that node thinks they are), so the mean number of chosen tips is $\frac{2r}{r+\lambda h}$. The key observation is now that, in the stationary regime, this mean number of chosen tips should be equal to 1, since, in average, a new coming transaction should not change the number of tips. Solving the equation $\frac{2r}{r+\lambda h} = 1$ with respect to r, we obtain r = λh, and so

$$L_0 = 2\lambda h \quad (1)$$

We also note that, if the rule is that a new transaction references k transactions instead of 2, then a similar calculation gives

$$L_0(k) = \frac{k\lambda h}{k-1} \quad (2)$$

This is, of course, consistent with the fact that $L_0(k)$ should tend to λh as $k \to \infty$ (basically, the only tips would be those still unknown to the network). Also (we return to the case of two transactions to approve) the expected time for a transaction to be approved for the first time is approximately $h + \frac{L_0}{2\lambda} = 2h$. This is because, by our assumption, during the first h units of time a transaction cannot be approved, and after that the Poisson flow of approvals to it has rate approximately $\frac{2\lambda}{L_0}$.

Observe that at any fixed time t the set of transactions that were tips at some moment $s \in [t; t + h(L_0, N)]$ typically constitutes a cutset. Any path from a transaction issued at time $t' > t$ to the genesis must pass through this set.

It is important that the size of a new cutset in the hive occasionally becomes small. One may then use the small cutsets as checkpoints for possible DAG pruning and other tasks. It is important to observe that the above "purely random" approval strategy is not very good in practice because it does not encourage approving tips. A "lazy" user could always approve a fixed pair of very old transactions, therefore not contributing to the approval of more recent transactions, without being punished for such behavior.

Also, a malicious entity can artificially inflate the number of tips by issuing many transactions that approve a fixed pair of transactions. This would make it possible for future transactions to select these tips with very high probability, effectively abandoning the tips belonging to "honest" nodes. To avoid issues of this sort, one has to adopt a strategy that is biased towards the "better" tips.

Before starting the discussion about the expected time for a transaction to receive its first approval, note that we can distinguish two regimes.

- Low load: the typical number of tips is small, and frequently becomes 1. This may happen when the flow of transactions is so small that it is not probable that several different transactions approve the same tip. Also, if the network latency is very low and devices compute fast, it is unlikely that many tips would appear. This even holds true in the case when the flow of transactions is reasonably large. Moreover, we have to assume that there are no attackers that try to artificially inflate the number of tips.
- High load: the typical number of tips is large. This may happen when the flow of transactions is large, and computational delays together with network latency make it likely that several different transactions approve the same tip.

This division is rather informal, and there is no clear borderline between the two regimes. Nevertheless, we find that it may be instructive to consider these two different extremes.

The situation in the low load regime is relatively simple. The first approval happens on an average timescale of order $\lambda - 1$ since one of the first few incoming transactions will approve a given tip.

Let us now consider the high load regime, the case where L0 is large. As mentioned above, one may assume that the Poisson flows of approvals to different tips are independent and have an approximate rate of $\frac{2\lambda}{L_0}$. Therefore, the expected time for a transaction to receive its first approval is around $L_0/(2\lambda) \approx 1.45h$ (1).

However, it is worth noting that for more elaborate approval strategies, it may not be a good idea to passively wait a long time until a transaction is approved by the others. This is due to the fact that "better" tips will keep appearing and will be preferred for approval.

Rather, in the case when a transaction is waiting for approval over a time interval much larger than $L_0/(2\lambda)$, a good strategy would be to promote this latent transaction with an additional empty transaction.

POOLS

To speed up the process of approving transactions, the system has group of nodes that are engaged in joint validation. It works like this: when a node is connected to the system, the network tells it which group it belongs to. After that, the time synchronization of all pool members takes place. At certain intervals, the network selects the most important transaction and report it to the pool, after that, each member of the pool begins confirmation. Let us explain, let n be the number of pool members, d the average number of iterations required for finding the nonce, i is the index of the pool member. Then, r = [(d / n) i; (d / n) i + d) the range of values that the i-th member of the pool needs to go over.

PROOF – OF – ME

This is a principle based on the Proof-Of-Work and the rating system. This protection principle assumes, that the user who wants to interact with the system, must first confirm himself.

When you add a new transaction to the graph, it confirms the two past ones. Transactions for confirmation are selected by a certain algorithm, which checks whether these transactions are not contredict and whether they are not accept conflict transactions. For further use, the proof of work is similar to that of Adam Beck's HashCash. The work on proving the reliability of transactions involves scanning to a value that, when hashed using the digest384 algorithm, starts with a certain number of zero trithes.

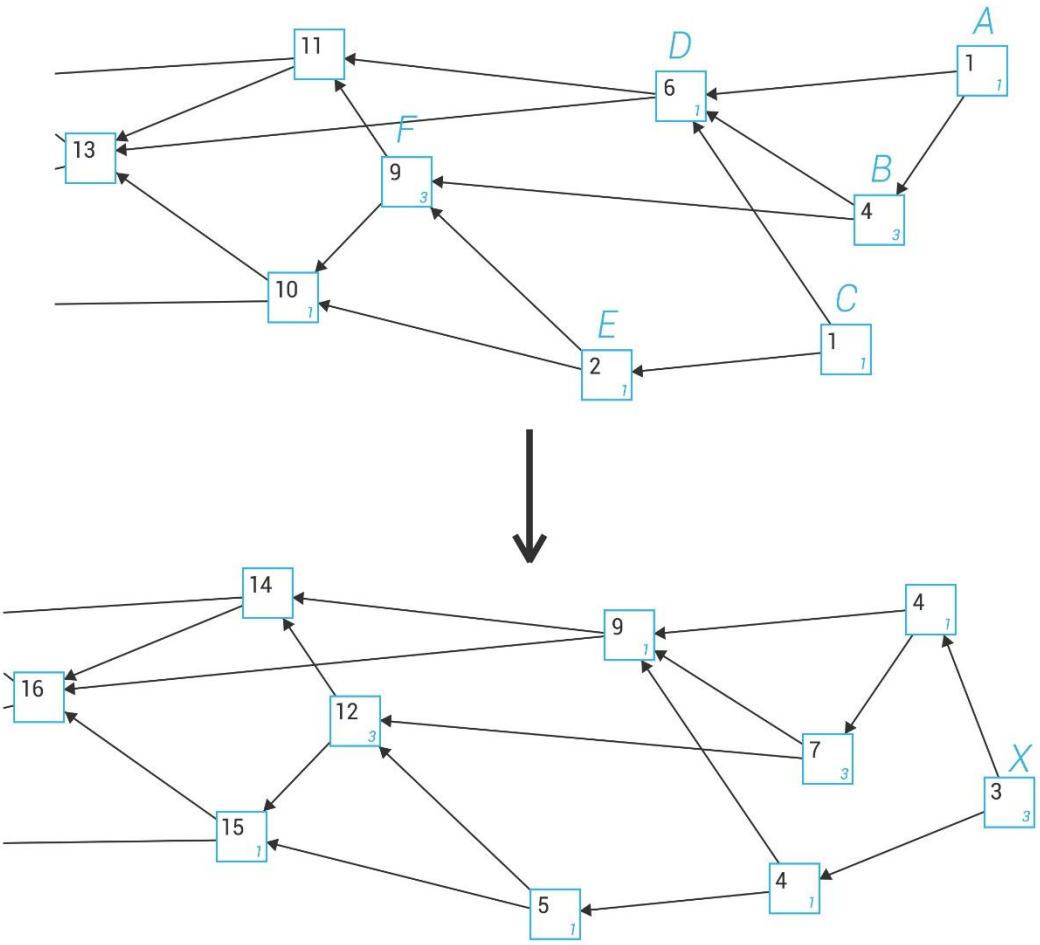

Figure 2. Example of adding a new transaction

The algorithm for calculating the total weight can be seen in the figure. Each node (square) is a transaction, the number at the bottom is the transaction's own weight, the number allocated by the bold-cumulative weight. In Figure 2, transaction F is directly or indirectly confirmed by transactions A, B, C, E. The total mass F is 9 (1 + 3 + 1 + 1 + 3). Transactions A, C are the ends of the graph. X, in the second picture, as they indirectly confirm, 3.

The hash is found and the weights are counted on the one with which the transaction was transferred, thereby the device itself is a miner.

In order for users to have an incentive to actively use the Hive, there is a rating system. Having counted the current ratings of the node, you can identify its activity, and depending on this, give more, or store preferences in confirming the transactions of this node. It works like this: a node does a certain job when making or confirming a transaction. It can be said that this work is recorded in the Hive in the form of transactions. Given the time stamps of each transaction, you can create an algorithm that will calculate the current node rating. Here is an example of one of such algorithms. Take the set of all the transactions of the node for a period of time. We assume that for one transaction the node rating is increased by C, whereas for one transaction, the rating is reduced by F. Note that the rating cannot be negative. Let C be the number of the rating for the transaction, D is the ordered set of such elements t-t_c, including the element 0, where t is the transaction timestamp (in days), t_c is the current timestamp (in days). Then the node's rating at the current time.

$R = \sum_{i=1}^{n-1} max\{R + C + F(D_i - D_{i+1}),0\}$, где $n = |D|$.

EXCHANGE

"Atomic Swap" (atomic swap) is the exchange of cryptocurrencies directly between two participants without the participation of a third party.

Let's say that Alice and Bob want to exchange one cryptocurrency to another. Alice transfers her funds to a kind of depository, in which the funds for the exchange will be kept until the end of the transaction. To withdraw funds from this cell, you need a secret key and Bob's signature.

Alice generates the private key and its hash. Then Bob asks Alice for this secret key and creates a similar cell to store his funds with the same key. Note that, as in the case of Alice's cell, Bob cannot open his cell without Alice's signature. At the same time, at this stage, Alice already has the opportunity to open Bob's cell by signing it, and get the money to her account. When Alice got the money, Bob gets her signature, with which he can open the second cell and complete the exchange.

In the event that one of the participants terminates the deal halfway, the atomic swap rejects the deal and returns the funds back to both participants. Smart contract (English Smart contract) is just required for making similar transactions. Smart contracts are stored in the Hive on the same principle as transactions, and in essence represent a bytecode that runs on a Paymon virtual machine (PVM).

The PVM is a simple stack-based architecture. The word size of the machine (and thus size of stack item) is 256-bit. This was chosen to facilitate the Keccak-256 hash scheme and elliptic-curve computations. The memory model is a simple word-addressed byte array. The stack has a maximum size of 1024. The machine also has an independent storage model; this is similar in concept to the memory but rather than a byte array, it is a word-addressable word array. Unlike memory, which is volatile, storage is nonvolatile and is maintained as part of the system state. All locations in both storage and memory

are well-defined initially as zero.

The machine does not follow the standard von Neumann architecture. Rather than storing program code in generally-accessible memory or storage, it is stored separately in a virtual ROM interactable only through a specialised instruction.

The machine can have exceptional execution for several reasons, including stack under ows and invalid instructions.

They do not leave state changes intact. Rather, the machine halts immediately and reports the issue to the execution agent (either the transaction processor or, recursively, the spawning execution environment) which will deal with it separately.

In addition to the system state, and the remaining gas for computation t, there are several pieces of important information used in the execution environment that the execution agent must provide; these are contained in the tuple I:

- Ia, the address of the account which owns the code that is executing.
- Io, the sender address of the transaction that originated this execution.
- Id, the byte array that is the input data to this execution; if the execution agent is a transaction, this would be the transaction data.
- Is, the address of the account which caused the code to be executing; if the execution agent is a transaction, this would be the transaction sender.
- Iv, the value, in Wei, passed to this account as part of the same procedure as execution; if the execution agent is a transaction, this would be the transaction value.
- Ib, the byte array that is the machine code to be executed.

The execution model denes the function P, which can compute the resultant state 0, the remaining gas g0, the suicide list s, the log series l, the refunds r and the resultant output, o, given these denfitions: $(\sigma', t', s, l, r, o) \equiv P(\sigma, g, I)$.

We must now dene the function P. In most practical implementations this will be modelled as an iterative progression of the pair comprising the full system state, σ and the machine state, μ. Formally, we dene it recursively with a function X. This uses an iterator function O (which denes the result of a single cycle of the state machine) together with functions Z which determines if the present state is an exceptional halting state of the machine and H, specifying the output data of the instruction if and only if the present state is a normal halting state of the machine.

The empty sequence, denoted (), is not equal to the empty set, denoted ∅; this is important when interpreting the output of H, which evaluates to ∅ when execution is to continue but a series (potentially empty) when execution should halt.

$$P(\sigma, t, I) \equiv X_{0,1,2,4}((\sigma, \mu, A^0, I))$$

$$\mu_g \equiv t$$

$$\mu_{pc} \equiv 0$$

$$\mu_m \equiv (0, 0, \ldots)$$

$$\mu_i \equiv 0$$

$$\mu_s \equiv ()$$

$$\mathrm{X}(\sigma, \mu, \mathrm{A}, \mathrm{I}) \equiv \begin{cases} (\emptyset, \mu, A^0, \mathrm{I}, ()), & \text{if } \mathrm{Z}(\sigma, \mu, \mathrm{I}) \\ \mathrm{O}(\sigma, \mu, \mathrm{A}, \mathrm{I}) \cdot \mathrm{o}, & \text{if } \mathrm{o} \neq \emptyset \\ \mathrm{X}\big(\mathrm{O}(\sigma, \mu, \mathrm{A}, \mathrm{I})\big), & \text{otherwise} \end{cases}$$

where

$$o \equiv H(\mu, I)$$

$$(a, b, c) \cdot d \equiv (a, b, c, d)$$

Note that we must drop the fourth value in the tuple returned by X to correctly evaluate P, hence the subscript X0;1;2;4.

X is thus cycled (recursively here, but implementations are generally expected to use a simple iterative loop) until either Z becomes true indicating that the present state is exceptional and that the machine must be halted and any changes discarded or until H becomes a series (rather than the empty set) indicating that the machine has reached a controlled halt.

BUSINESS PLATFORM PROFIT

The system has a business platform, through which you can easily connect your own store, service, etc. and accept payments for goods or services in the cryptocurrency. On the site there is a handy designer, who after the addition of goods / services generates smart contracts. These smart contracts provide the work of the platform.

For example, when a user selects a product or service and clicks to buy, he has a choice, show a QR code and pay for the goods on the spot, or choose delivery. In the second case, the smart contract will wait until the company confirms the purchase, checks whether the user has sufficient funds, and makes an exchange.

In order to avoid conflicts between the client and the company, it is rated. Each user, using the services of the company, can put an estimate from 1 to 5. Also, you can always open a chat with this company and discuss the question. Also later, due to the created smart contracts inside Hive and Profit, all our users can conduct their own ICO and arrange collective fees for any needs. After all, Paymon does not have a commission, and transactions have the lightest weight in comparison with competitors.

Let's consider in detail the question of why we need a cashback and where it comes from. Cashback is a kind of reward for the use of services, therefore, it will be taken from the account of this company. Depending on the rating, the number of paid tokens (cashback) will be equal.

$$M = \begin{cases} \mathrm{A} * 0.0005, \text{if } \mathrm{R} > 10 \wedge \mathrm{R} < 25 \\ \mathrm{A} * 0.0010, \text{if } \mathrm{R} \geq 25 \wedge \mathrm{R} < 50 \\ \mathrm{A} * 0.0015, \text{if } \mathrm{R} \geq 50 \wedge \mathrm{R} < 75 \\ \mathrm{A} * 0.0020, \quad \text{otherwise} \end{cases}$$