AICHAIN Yellowpaper

Version: 9.1

1. Abstract

The aim of AICHAIN is to provide a public blockchain platform for the AI with complex applications. The data resources provider, application development team and runtime platform resources provider and users, they all can deploy the data or application on AICHAIN, and setup an ecology chain system for AI application on blockchain technology with lower cost and lower technology threshold.

AICHAIN will build a benign ecosystem, encourage more people to participate in the development and landing of AI applications, promote the development of AI in a credible and reliable environment, and transform the data generated by individuals into more precise services for individuals.

The key issues to be solved by the AICHAIN:

1. Based on the blockchain technology of bitcoin and implement a non-chip mining algorithm. Bitcoin-HASH, LTC-scrypt, DASH-X11 have been implemented inside ASIC. ETH is to take up a lot of graphics resources. High-end graphics cards are now also monopolized by big companies. A less expensive graphics card resource is expected, but at the same time it must not be easily implemented inside ASIC). Its purpose is to make blockchain safer, to ensure that participating users have enough the computing power and the right to deploy their own applications.

2. Develop AI application deployment functions based on bitcoin blockchain and definitions of the application deployment or execution unit based on the data format of the transaction.

3. AICHAIN separates the application running environment from the blockchain node. Use docker as the application running platform, which allows AICHAIN to provide standard, upgradeable, customizable and supporting multiple programming languages APP running environment. AICHAIN node program comes with a public standard docker IMG with application running environment. This running environment can be continuously upgraded, even be modified by user freely, and users can deploy their own docker IMG with AICHAIN nodes.

4. Only put the description of AI application on the blockchain, and will not include complete application data, which allows that the size of application data can be very large, and can save the storage space of blockchain. The deployer provides download address for application executable file and data resources, or an address for direct service. Only the description information of those application or resource is recorded in the AI application information unit.

5. Provide the interface for verification of user's identity and blockchain transaction information. This allows developers make more customized on their application, which is suitable for more complex application development and running environment.

2. Definition of private key, public key and address

2.1 Public Key Cryptography

Asymmetric encryption requires two (one pair) keys: public key and private key. Only the corresponding private key can decrypted the data encrypted with the public key. And vice versa. (If the private key is used in Encryption, only the corresponding public key can be used to decrypt it.) Both parties can establish secure communication without exchanging keys.

The AICHAIN system uses the same specification in Bitcoin, the private key is composed of 32 bytes of random numbers, the public key can be calculated by the private key, and the public key obtains the address of the AICHAIN's coin through a series of hashing and encoding algorithms. So the address is actually another form of public key, it can be interpreted as a summary of the public key.

2.2 Related Algorithms

There are several algorithms used in the calculation of private key, public key and address, such as a signature algorithm based on secp256k1 elliptic curve multiplication, SHA-256, RIPEMD-160, and Base58 encoding.

2.2.1 Elliptic Curve Signature Algorithm

The use of elliptic curves in cryptography was independently proposed by Neal Koblitz and Victor Miller in 1985, respectively. Its main advantage is that in some cases it uses smaller keys than other algorithms (like RSA) but provides comparable or higher level of security.

Bitcoin uses a public-key cryptography algorithm based on secp256k1 elliptic curve mathematics. It contains a private key and a public key. The private key is used to sign the transaction, and send the signature and the original data the entire virtual coin network. The public key is used by the nodes in the whole network to verify the validity of the transaction. The signature algorithm ensures that the transaction is issued by the person who owns the corresponding private key.

2.2.2 Hash Function

SHA-256 is a kind of hash function.

RIPEMD-160 is also a kind of hash function used to get address, with an output of 20 bytes (160 bits). Bitcoin uses it to reduce the number of bytes of the mark that receiver get.

2.2.3 Base58 encoding

It is a kind of readability coding algorithm, similar to the replacement algorithm in classical cryptography. It isn’t the core of cryptography in theory. Readability coding algorithm is not for data security, but for readability. Information transmitted in binary is not readable, the strings composed of numbers and letters are more easily identified. Readability coding does not change the content of information, only change the form of it (some coding algorithm also incorporates fault-tolerant parity function to ensure the accuracy and completeness of the data during transmission).

Base64 is a common readability encoding algorithm. The name of this algorithm means that 64 characters are used in the encoding process: uppercase A to Z, lowercase a to z, numbers 0 to 9, "+" and "/".

Base58 is a coding method used in Bitcoin, and mainly used to generate Bitcoin wallet address. Comparing to Base64, Base 58 does not use the number "0", the capital "O", the capital "I", and the lowercase "i", as well as the "+" and "/" symbols.

The main purpose of design Base58 is :

Avoid confusion. Under some fonts, the number 0 and the capital letter O, as well as the capital letter I and the letter lower case l, are very similar.

The reason why "+" and "/" are not used is that the characters which are neither numbers nor letters are difficult to accept as part of an account.

No punctuation. It usually not separate from the middle.

To make most of the software support double-click to select the entire string.

AICHAIN’s coin also uses the Base58 algorithm to encode the hash 160 of public key and private key, generate the address starting with A and the private key in WIF (Wallet import Format) format.

2.3 Private Key and Public Key

The private key is actually a random number of 32 bytes (256 bits) generated by SHA-256. The range of valid private keys depends on the secp256k1 elliptic curve digital signature standard used by Bitcoin. Almost any number between 0x01 and 0xFFFF FFFF FFFF FFFF

FFFF FFFF FFFF FFFE BAAE DCE6 AF48 A03B BFD2 5E8C D036 4140 is a valid private key.

To get a common WIF (Wallet import Format)-format private key, you need to add version number in front of the key, compression flag and additional checksum after the key (Additional checksum is the first four bytes of the results of performing SHA-256 hash operation on the key twice.), and encode the key based on Base58 format.

After elliptic curve multiplication operation, you can get the public key from the private key. The public key is the point on the elliptic curve and has the x and y coordinates. There are two forms of public keys: compressed and uncompressed. Earlier bitcoin used an uncompressed public key, and most clients today use compressed public keys by default.

Due to the mathematical principle, it is feasible to derive the public key from the private key, but it is impossible to derive the private key backwards from the public key.

People who first hear bitcoin usually have a misconception that the bitcoin public key is just address, which is not correct. Because you need to go through some operations from the public key to the address.

2.4 Generating the address

The length of the generated public key based on elliptic curve algorithm is always too long: the compressed format has 33 bytes and non-compressed has 65 bytes. The address is to reduce the number of bytes that the receiver needs to identify. The address is generated as follows:

Generate the private key and public key.

Perform SHA256 hash algorithm on the public key to get a 32-byte hash value.

Perform RIPEMD-160 algorithm on the gained hash value to get a 20-byte hash value, which is called Hash160.

Perform SHA256 hash algorithm twice on the 21-byte array which is composed with version number and Hash160 to get the first four bytes of the result, which is called checksum. And put the gained checksum after the 21-byte array.

Encode the 25-byte array based on Base58 format and then get the address.

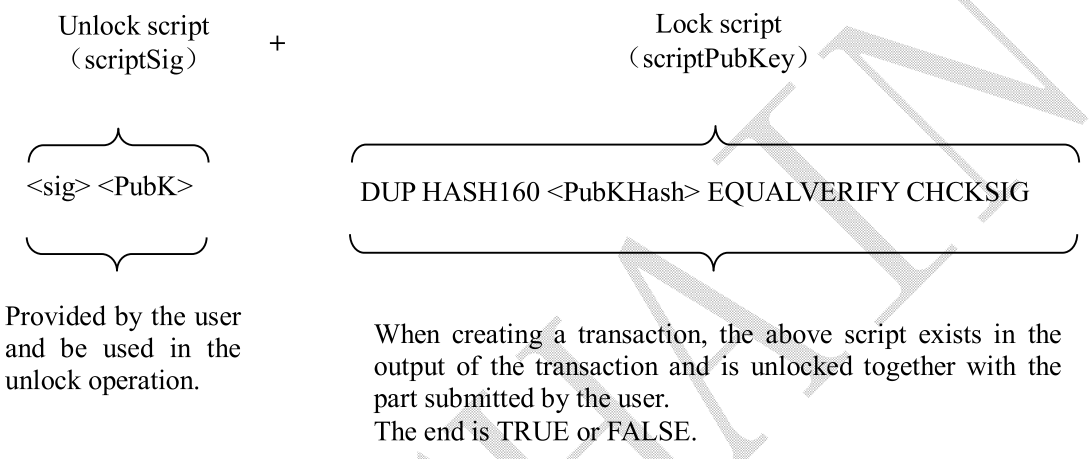
Due to the characteristics of elliptic curve multiplication and hash function, you can derive the public key from the private key and also derive the address from the public key, and this process is irreversible. Because of this, in the entire system, the private key is the most crucial part. Leaking private key means losing everything.

If you want to spend an asset on an address, you need to create a transaction and use the private key corresponding to that address to sign it. And if y want to transfer assets to one address, just need to transfer public address.

2.5 Some address forms

2.5.1 P2PKH（Pay to Public Key Hash）

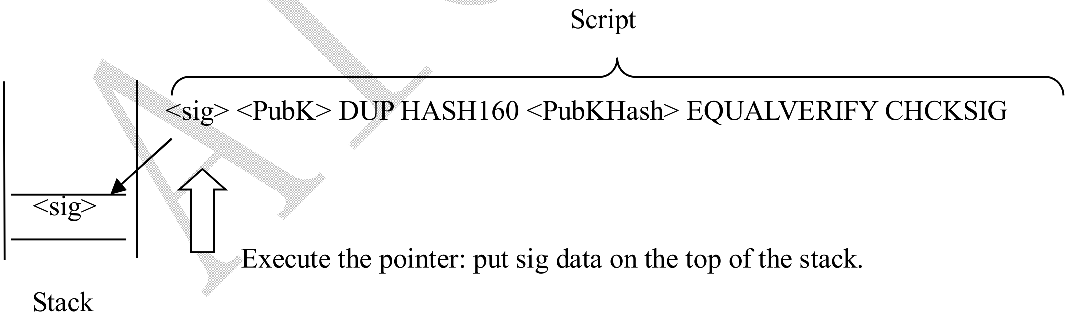
In present bitcoin network, most of transactions are based on P2PKH way. The following is a P2PKH lock script and unlock script:



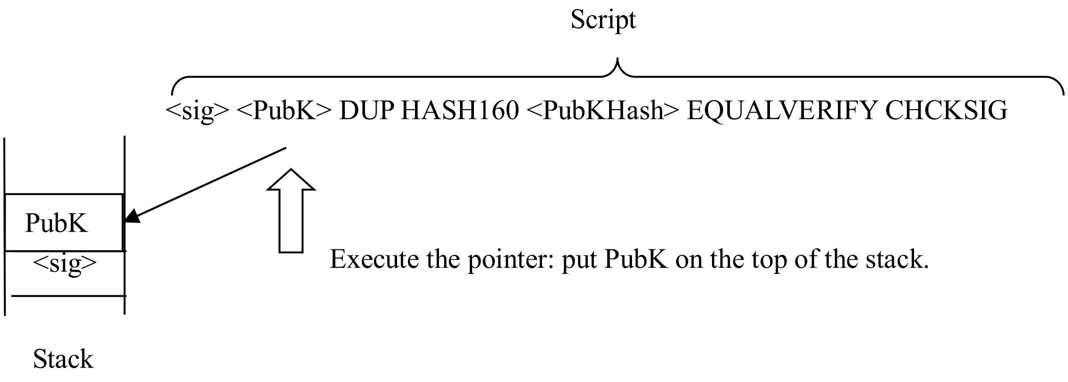
Provided by the user and be used in the unlock operation.

When creating a transaction, the above script exists in the output of the transaction and is unlocked together with the part submitted by the user.
The end is TRUE or FALSE.

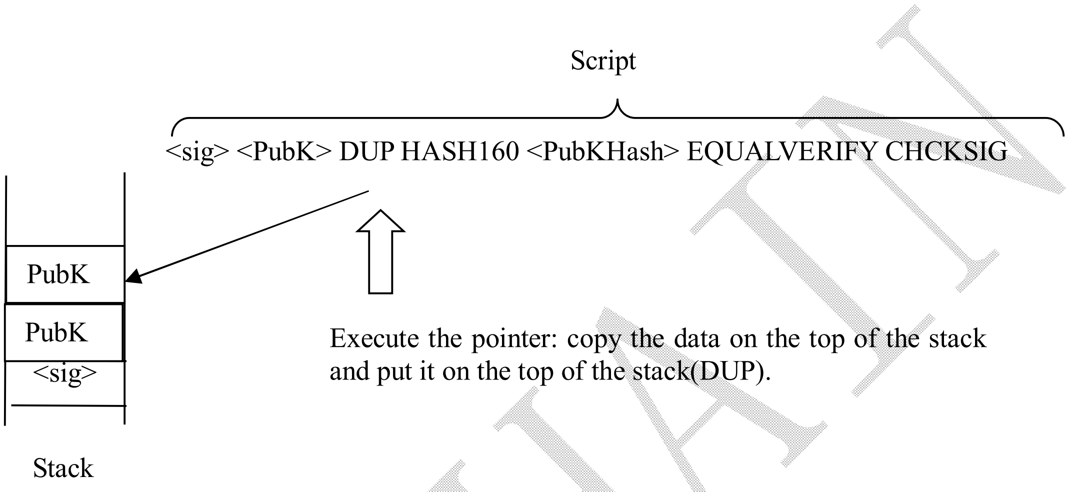
<sig> means the signature, and <PubK> means the public key. And specific steps are as follows:



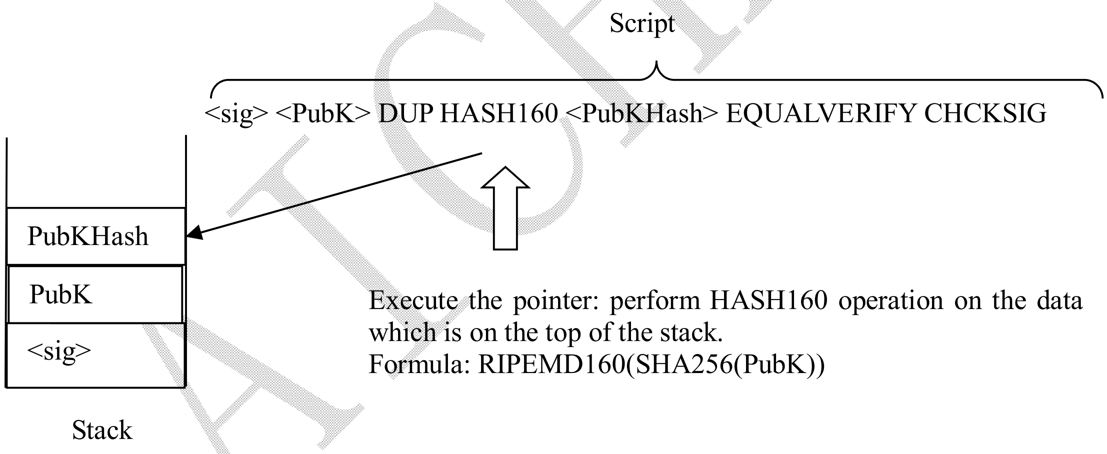
Execute the pointer: put sig data on the top of the stack.



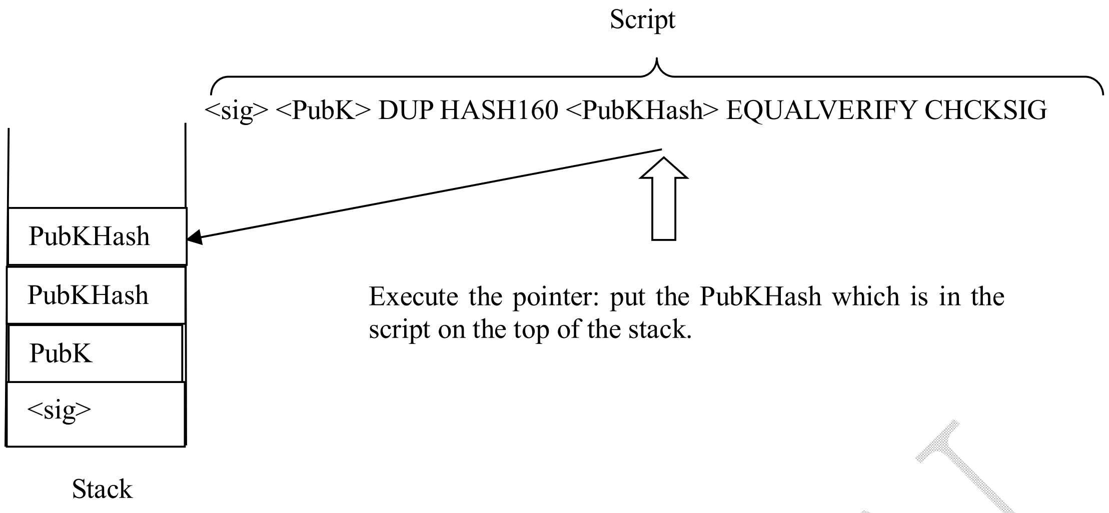
Execute the pointer: put PubK on the top of the stack.



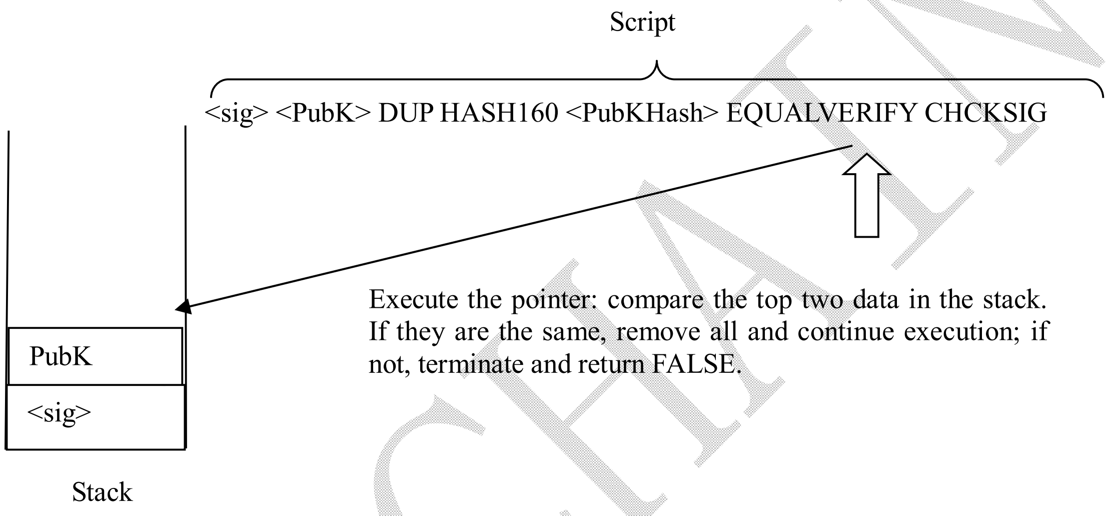
Execute the pointer: copy the data on the top of the stack and put it on the top of the stack(DUP).



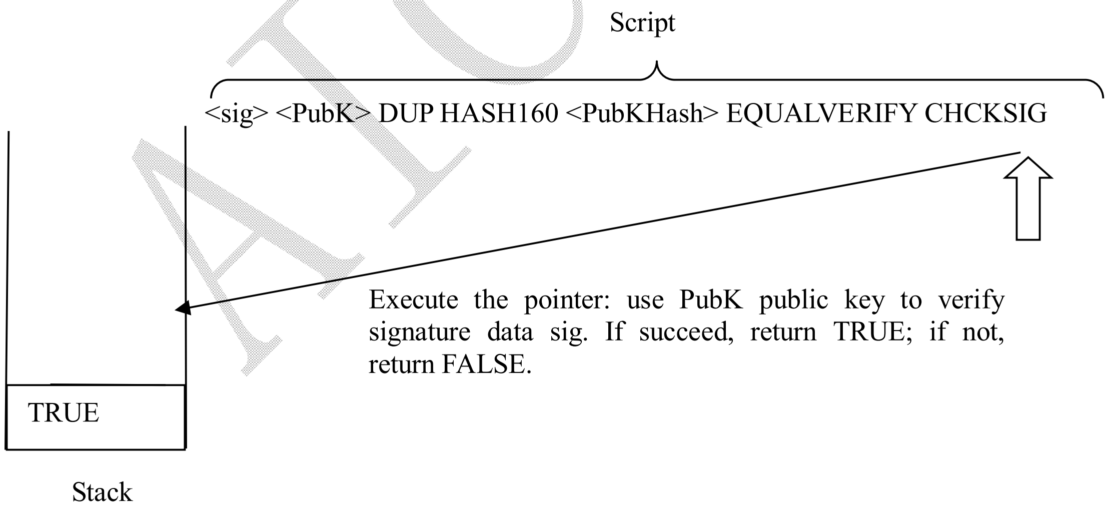
Execute the pointer: perform HASH160 operation on the data which is on the top of the stack.
Formula: RIPEMD160(SHA256(PubK))



Execute the pointer: put the PubKHash which is in the script on the top of the stack.



Execute the pointer: compare the top two data in the stack. If they are the same, remove all and continue execution; if not, terminate and return FALSE.



Execute the pointer: use PubK public key to verify signature data sig. If succeed, return TRUE; if not, return FALSE.

It can be seen that two tests are mainly verified, the first is whether Public Key can be converted to the correct address, and the second is whether Signature is right, that is, whether you are the owner of Public Key, that is, you have the corresponding private key (Private Key).

The content of Signature is mainly the result of operations between the transaction (the

Hash of transaction information) and the private key , which is usually a coordinate data R, S. At the time of verification, the signature results, the transaction summary, and the public key are performed, and the result of a verification signature is finally obtained: TRUE or FALSE.

2.5.2 P2PK（Pay to Public Key）

The P2PK lock version of the script is as follows:

<Public Key A> OP_CHECKSIG

The script used for unlocking is a simple signature:

<Signature from Private Key A>

The combination script confirmed by the transaction validation software is:

<Signature from Private Key A> <Public Key A> OP_CHECKSIG

It is found that this rule is much simpler than P2PKH, and only one step is verified and the above address verification is less. In fact, the main creation purpose of P2PKH is to make the address shorter and make it more convenient to use. What's more, the core content of Bitcoin is also the P2PK.

2.5.3 P2SH（Pay to Script Hash）

The general form of the locking script with M-N multisignature is:

M <Public Key 1> <Public Key 2> ... <Public Key N> N OP_CHECKMULTISIG

Among them, N is the total number of archived public keys, and M is the minimum public key that requires the activation of the transaction.

For example, 2-3 multisignature conditions:

2 <Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG

The above locked script can be unlocked by a script containing a signature and a public key:

OP_0 <Signature B> <Signature C>

OP_0 is a placeholder, and it doesn't have any practical significance.

Two combinations of scripts will form a validation script:

OP_0 <Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG

P2SH is a simplified version of the MS multisignature. If P2SH is used to do the same 2-3 multiple signature conditions as above, the steps are as follows:

Lock a script:

2 <Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG

For locking scripts, the SHA256 hash algorithm is first used, and then the RIPEMD160 algorithm is applied to it. 20 bytes of one script:

8ac1d7a2fa204a16dc984fa81cfdf86a2a4e1731

So the lock of a script is changed to:

OP_HASH160 8ac1d7a2fa204a16dc984fa81cfdf86a2a4e1731 OP_EQUAL

This lock-in script is much shorter than the locking script originally used by MS. When the receivers want to use the UTXO in this transaction, they need to submit the unlocking script (here it can also be called the redemption script):

<Sig1> <Sig2> <2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG>

Combined with locking a script:

<Sig1> <Sig2> <2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG> OP_HASH160 8ac1d7a2fa204a16dc984fa81cfdf86a2a4e1731 OP_EQUAL

Using the operation rules in the 2.5.1 chapter, it is obvious that the verification process is divided into two steps. First, it is whether the redemption script attached to the receiver is consistent with the sender's locking script. If so, it will execute the script and verify the multisignature.

The feature of P2SH is that the responsibility of making the script is given to the recipient, the advantage of which is to postpone the pressure of the node's storage.

The multisignature address is that an address is generated by 2 or more public keys. When using N public keys to generate multiple signature addresses, it can be agreed that at least M private keys are needed to sign the transaction when verifying the signatures. Meet the following conditions:

N >= 2

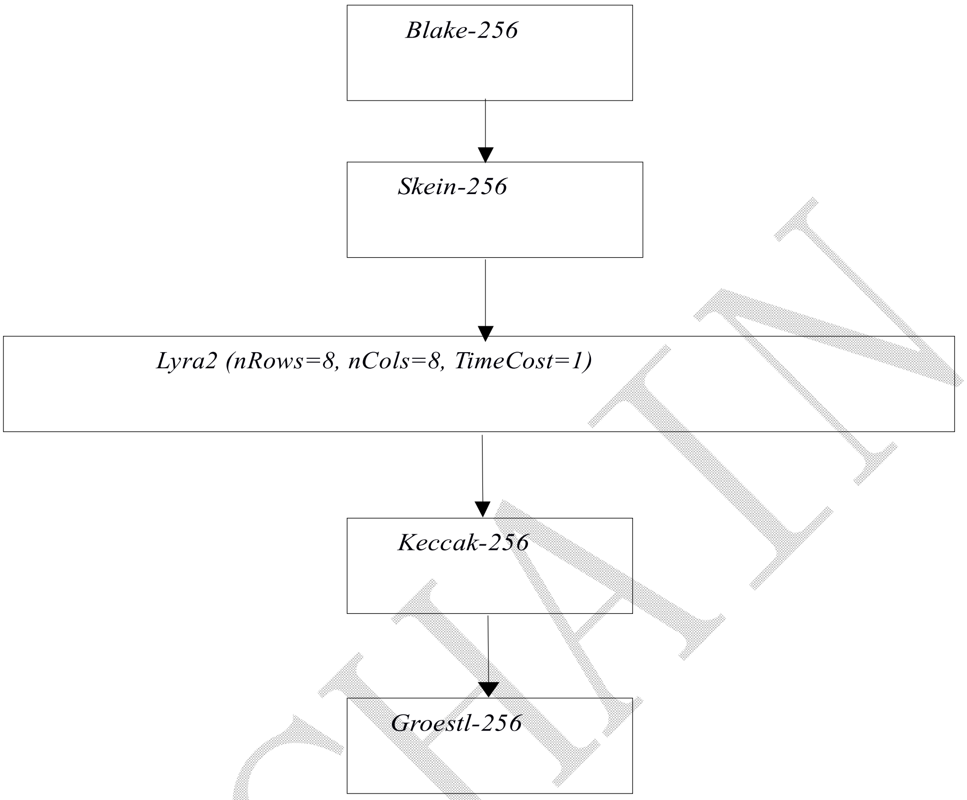
N >= M >= 1

3. Mining Algorithm

3.1 Lyra2DC (DC – Dynamic Complexity)

A NIST5 based chained algorithm “Lyra2DC” (DC – Dynamic Complexity), is proposed with customizable parameters useful for thwarting future ASIC (Application Specific Integrated Circuit) threats. Adapted from the Lyra2RE algorithm used by Vertcoin and Monacoin, Lyra2DC is specifically designed with this purpose in mind affording lower power consumption and cooler GPU temperatures. Lyra2 (the principal part of the chained algorithm) allows us to change memory usage and time cost independently, giving us more leverage against ASICs.

Lyra2DC is a chained algorithm consisting of five different hash functions: Keccak, Skein, Groestl, Blake and Lyra2.



Leveraging industry proven hashing algorithms, we were able to create the most secure, robust, enduring chained algorithm to date that is both easier on GPUs and resistant to ASICs. At this time we have decided not to implement an “N factor” schedule as it is nearly impossible to predict the future. However, Lyra2DC will give us the flexibility to make changes whenever that becomes necessary.

Due to the chained nature of the algorithm, GPU miners will be inherently hard to optimize, meaning that power draw and heat can be reduced. This has been a desired feature for some time with Scrypt-N coins seeing dropping hash rates due to high energy consumption, while Vertcoin (which used Lyra2RE) having consistently the highest $/Day/Normalized MH/s than other coins.

As was previously detailed in the Lyra2 white paper, Lyra2 is strictly sequential in nature, using a “cryptographic sponge” at its core. This means that parallelization of the algorithm will be practically impossible with each step relying on the previous step having already been computed.

Unlike Scrypt-N, time cost and memory cost are separated, giving us independent control over both parameters. ASICs have been far easier to develop for Scrypt-N than they will be for Lyra2DC because increasing the N-factor of Scrypt simply involves doing more iterations of the algorithm. Under Lyra2, whilst increasing the time cost only involves more iteration, increasing the memory requirement means that any potential ASIC device would have to physically be designed with more memory for each thread. In the future, if ASICs ever were developed for Lyra2DC, we would simply have to fork to a higher memory requirement and those ASICs would no longer properly function.

Many crypto-currencies claim to have ASIC-resistant algorithms, but many of them are only so because no ASIC has been made for them yet. It has been rumored that FPGAs for X11 already exist and Neoscrypt only uses more rounds of cipher functions. By contrast, Lyra2DC aims to be ASIC-resistant at heart, allowing for less disruption to miners in the future due to our ability to change algorithm parameters rather than change algorithm all together. It will also free up development time to focus on new features without having to worry about constantly implementing new algorithms every time there is an ASIC threat.

3.2 Password Hashing Schemes (PHS)

As previously discussed, the basic requirement for a PHS is to be non-invertible, so that recovering the password from its output is computationally infeasible. Moreover, a good PHS's output is expected to be indistinguishable from random bit strings, preventing an attacker from discarding part of the password space based on perceived patterns. In principle, those requirements can be easily accomplished simply by using a secure hash function, which by itself ensures that the best attack venue against the derived key is through brute force (possibly aided by a dictionary or "usual" password structures).

What any modern PHS do, then, is to include techniques that raise the cost of brute-force attacks. A first strategy for accomplishing this is to take as input not only the user memorizable password pwd itself, but also a sequence of random bits known as salt. The presence of such random variable thwarts several attacks based on pre-built tables of common passwords, i.e., the attacker is forced to create a new table from scratch for every different salt. The salt can, thus, be seen as an index into a large set of possible keys derived from pwd, and need not to be memorized or kept secret.

A second strategy is to purposely raise the cost of every password guess in terms of computational resources, such as processing time and/or memory usage. This certainly also raises the cost of authenticating a legitimate user entering the correct password, meaning that the algorithm needs to be configured so that the burden placed on the target platform is minimally noticeable by humans. Therefore, the legitimate users and their platforms are ultimately what impose an upper limit on how computationally expensive the PHS can be for themselves and for attackers. For example, a human user running a single PHS instance is unlikely to consider a nuisance that the password hashing process takes 1 s to run and uses a small part of the machine's free memory, e.g., 20 MB. On the other hand, supposing that the password hashing process cannot be divided into smaller parallelizable tasks, achieving a throughput of 1,000 passwords tested per second requires 20 GB of memory and 1,000 processing units as powerful as that of the legitimate user.

A third strategy, especially useful when the PHS involves both processing time and memory usage, is to use a design with low parallelizability. The reasoning is as follows. For an attacker with access to p processing cores, there is usually no difference between assigning one password guess to each core or parallelizing a single guess so it is processed p times faster: in both scenarios, the total password guessing throughput is the same. However, a sequential design that involves configurable memory usage imposes an interesting penalty to attackers who do not have enough memory for running the p guesses in parallel. For example, suppose that testing a guess involves m bytes of memory and the execution of n instructions. Suppose also that the attacker's device has 100m bytes of memory and 1000 cores, and that each core executes n instructions per second. In this scenario, up to 100 guesses can be tested per second against a strictly sequential algorithm (one per core), the other 900 cores remaining idle because they have no memory to run.

Aiming to provide a deeper understanding on the challenges faced by PHS solutions, in what follows we discuss the main characteristics of platforms used by attackers and then how existing solutions avoid those threats.

- Attack platforms：

The most dangerous threats faced by any PHS comes from platforms that benefit from "economies of scale", especially when cheap, massively parallel hardware is available. The most prominent examples of such platforms are Graphics Processing Units (GPUs) and custom hardware synthesized from FPGAs.

1. Graphics Processing Units (GPUs). Following the increasing demand for high-definition realtime rendering, Graphics Processing Units (GPUs) have traditionally carried a large number of processing cores, boosting its parallelization capability. Only more recently, however, GPUs evolved from specific platforms into devices for universal computation and started to give support to standardized languages that help harness their computational power, such as CUDA and OpenCL). As a result, they became more intensively employed for more general purposes, including password cracking.

As modern GPUs include a few thousands processing cores in a single piece of equipment, the task of executing multiple threads in parallel becomes simple and cheap. They are, thus, ideal when the goal is to test multiple passwords independently or to parallelize a PHS's internal instructions. For example, NVidia's Tesla K20X, one of the top GPUs available, has a total of 2,688 processing cores operating at 732 MHz, as well as 6 GB of shared DRAM with a bandwidth of 250 GB per second. Its computational power can also be further expanded by using the host machine's resources, although this is also likely to limit the memory throughput. Supposing this GPU is used to attack a PHS whose parametrization makes it run in 1 s and take less than 2.23 MB of memory, it is easy to conceive an implementation that tests 2,688 passwords per second. With a higher memory usage, however, this number is deemed to drop due to

the GPU's memory limit of 6 GB. For example, if a sequential PHS requires 20 MB of DRAM, the maximum number of cores that could be used simultaneously becomes 300, only 11% of the total available.

2． Field Programmable Gate Arrays (FPGAs). An FPGA is a collection of configurable logic blocks wired together and with memory elements, forming a programmable and high-performance integrated circuit. In addition, as such devices are configured to perform a specific task, they can be highly optimized for its purpose (e.g., using pipelining). Hence, as long as enough resources (i.e., logic gates and memory) are available in the underlying hardware, FPGAs potentially yield a more cost-effective solution than what would be achieved with a general-purpose CPU of similar cost. When compared to GPUs, FPGAs may also be advantageous due to the latter's considerably lower energy consumptio, which can be further reduced if its circuit is synthesized in the form of custom logic hardware (ASIC).

A recent example of password cracking using FPGAs is presented in. Using a RIVYERA S3-5000 cluster with 128 FPGAs against PBKDF2- SHA-512, the authors reported a throughput of 356,352 passwords tested per second in an architecture having 5,376 password processed in parallel. It is interesting to notice that one of the reasons that made these results possible is the small memory usage of the PBKDF2 algorithm, as most of the underlying SHA-2 processing is performed using the device's memory cache (much faster than DRAM). Against a PHS requiring 20 MB to run, for example, the resulting throughput would presumably be much lower, especially considering that the FPGAs employed can have up to 64 MB of DRAM and, thus, up to three passwords can be processed in parallel rather than 5,376.

Interestingly, a PHS that requires a similar memory usage would be troublesome even for state-of-the-art clusters, such as the newer RIVYERA V7-2000T. This powerful cluster carries up to four Xilinx Virtex-7 FPGAs and up to 128 GB of shared DRAM, in addition to the 20 GB available in each FPGA. Despite being much more powerful, in principle it would still be unable to test more than 2,600 passwords in parallel against a PHS that strictly requires 20MB to run.

3.3 Scrypt

Now we briefly describe the Scrypt and Lyra2 algorithm for the sake of completeness. Scrypt was a 2nd generation POW algorithm, compared with the RSA256 used in bitcoin, while we believe the Lyra2 is a third generation algorithm for improved asic resistance.

Arguably, the main password hashing solutions available in the literature are: PBKDF2, bcrypt and scrypt. Since scrypt is only PHS among them that explores both memory and processing costs and, thus, is directly comparable to Lyra2, its main characteristics are described in what follows. For the interested reader, a discussion on PBKDF2 and bcrypt is provided in the appendices.

The design of scrypt focus on coupling memory and time costs. For this, scrypt employs the concept of "sequential memory-hard" functions: an algorithm that asymptotically uses almost as much memory as it requires operations and for which a parallel implementation cannot asymptotically obtain a significantly lower cost. As a consequence, if the number of operations and the amount of memory used in the regular operation of the algorithm are both (R), the complexity of a memory-free attack (i.e., an attack for which the memory usage is reduced to (1)) becomes Ω(R2), where R is a system parameter. We refer the reader to for a more formal definition.

The following steps compose scrypt's operation (see Algorithm 1). First, it initializes p b-long memory blocks Bi. This is done using the PBKDF2 algorithm with HMAC-SHA-256 as underlying hash function and a single iteration. Then, each Bi is processed (incrementally or in parallel) by the sequential memory-hard ROMix function. Basically, ROMix initializes an array M of R b-long elements by iteratively hashing Bi. It then visits R positions of M at random, updating the internal state variable X during this (strictly sequential) process in order to ascertain that those positions are indeed available in memory. The hash function employed by ROMix is called BlockMix , which emulates a function having arbitrary (b-long) input and output lengths; this is done using

Algorithm 1 Scrypt.

```
Param: $h$   d $BlockMix$ 's internal hash function output length
Input: $pwd$ d The password
Input: $salt$ d A random salt
Input: $k$   d The key length
Input: $b$   d The block size, satisfying $b = 2r \cdot h$
Input: $R$   d Cost parameter (memory usage and processing time)
Input: $p$   d Parallelism parameter
Output: $K$  d The password-derived key
  $(B_0 ... B_{p-1}) \leftarrow$ PBKDF2$_{HMAC-SHA-256}$($pwd$, $salt$, 1, $p \cdot b$)
for $i \leftarrow 0$ to $p - 1$ do
    $B_i \leftarrow$ ROMix($B_i$, $R$)
end for
  $K \leftarrow$ PBKDF2$_{HMAC-SHA-256}$($pwd$, $B_0$ "$B_1$ "... "$B_{p-1}$, 1, $k$)
  return $K$   d Outputs the $k$-long key
  function ROMix($B$, $R$)   d Sequential memory-hard function
    $X \leftarrow B$
    for $i \leftarrow 0$ to $R - 1$ do   d Initializes memory array $M$
        $V_i \leftarrow X$ ; $X \leftarrow$ BlockMix($X$)
    end for
    for $i \leftarrow 0$ to $R - 1$ do   d Reads random positions of $M$
        $j \leftarrow Integerify(X)$ mod $R$
        $X \leftarrow$ BlockMix($X \oplus M_j$)
    end for
    return $X$
end function
function BlockMix($B$)   d $b$-long in/output hash function
    $Z \leftarrow B_{2r-1}$   d $r = b/2h$, where $h = 512$ for Salsa20/8
    for $i \leftarrow 0$ to $2r - 1$ do
        $Z \leftarrow Hash(Z \oplus B_i)$ ; $Y_i \leftarrow Z$
    end for
    return ($Y_0$, $Y_2$, ..., $Y_{2r-2}$, $Y_1$, $Y_3$, $Y_{2r-1}$)
end function
```

the Salsa20/8 stream cipher, whose output length is h = 512. After the p ROMix processes are over, the Bi blocks are used as salt in one final iteration of the PBKDF2 algorithm, outputting key K.

Scrypt displays a very interesting design, being one of the few existing solutions that allow the configuration of both processing and memory costs. One of its main shortcomings is probably the fact that it strongly couples' memory and processing requirements for a legitimate user. Specifically, scrypt's design prevents users from raising the algorithm's processing time while maintaining a fixed amount of memory usage, unless they are willing to raise the p parameter and allow further parallelism to be exploited by attackers. Another inconvenience with scrypt is the fact that it employs two different underlying hash functions, HMAC-SHA-256 (for the PBKDF2 algorithm) and Salsa20/8 (as the core of the BlockMix function), leading to increased implementation complexity. Finally, even though Salsa20/8's known vulnerabilities are not expected to put the security of scrypt in hazard, using a stronger alternative would be at least advisable, especially considering that the scheme's structure does not impose serious restrictions on the internal hash algorithm used by BlockMix. In this case, a sponge function could itself be an alternative, with the advantage that, since sponges support inputs and outputs of any length, the whole BlockMix structure could be replaced.

Inspired by scrypt's design, Lyra2 builds on the properties of sponges to provide not only a simpler, but also more secure solution. Indeed, Lyra2 stays on the "strong" side of the memory-hardness concept: the processing cost of attacks involving less memory than specified by the algorithm grows much faster than quadratic ally, surpassing the best achievable with scrypt and thwarting the exploitation of time-memory trade-offs (TMTO). This characteristic should discourage attackers from trading memory usage for processing time, which is exactly the goal of a PHS in which usage of both resources are configurable. In addition, Lyra2 allows for a higher memory usage for a similar processing time, increasing the cost of regular attack venues (i.e., those not exploring TMTO) beyond that of scrypt's.

3.4 Lyra2

As any PHS, Lyra2 takes as input a salt and a password, creating a pseudorandom output that can then be used as key material for cryptographic algorithms or as an authentication string. Internally, the scheme's memory is organized as a matrix that is expected to remain in memory during the whole password hashing process: since its cells are iteratively read and written, discarding a cell for saving memory leads to the need of re-computing it whenever it is accessed once again, until the point it was last modified. The construction and visitation of the matrix is done using a stateful combination of the absorbing, squeezing and duplexing operations of the underlying sponge (i.e., its internal state is never reset to zero), ensuring the sequential nature of the whole process. Also, the number of times the matrix's cells are revisited after initialization is defined

Algorithm 2 The Lyra2 Algorithm.

```
Param:  $H$  d Sponge with block size $b$ (in bits) and underlying permutation $f$ Param:
$H_\rho$ d Reduced-round sponge for use in the Setup and Wandering phases
Param:  $\omega$  d Number of bits to be used in rotations (recommended: a multiple of $W$ )
Input:  $pwd$  d The password
Input:  $salt$  d A salt
Input:  $T$  d Time cost, in number of iterations ($T$ “ 1)
Input:  $R$  d Number of rows in the memory matrix
Input:  $C$  d Number of columns in the memory matrix (recommended: $C \cdot \rho$ “ $\rho_{max}$)
Input:  $k$  d The desired hashing output length, in bits
Output:  $K$  d The password-derived $k$-long hash
d Bootstrapping phase: Initializes the sponge's state and local variables
d Byte representation of input parameters (others can be added)
$params \longleftarrow len(k)$ "$len(pwd)$ "$len(salt)$ "$T$ "$R$ "$C$
$H.absorb($pad$(pwd$ " $salt$ " $params))$  d Padding rule: 10*1.
$gap \longleftarrow 1$ ; $stp \longleftarrow 1$ ; $wnd \longleftarrow 2$ ; $sqrt \longleftarrow 2$  d Initializes visitation step and window
$prev^0 \longleftarrow 2$ ; $row^1 \longleftarrow 1$ ; $prev^1 \longleftarrow 0$
d Setup phase: Initializes a ($R \times C$) memory matrix, it's cells having $b$ bits each
for ($col \longleftarrow 0$ to $C - 1$) do {$M[0][C-1-col] \longleftarrow H_\rho.squeeze(b)$}end for
for ($col \longleftarrow 0$ to $C - 1$) do {$M[1][C-1-col] \longleftarrow M[0][col] \oplus H_\rho.duplex(M[0][col], b)$}end for
for ($col \longleftarrow 0$ to $C - 1$) do {$M[2][C-1-col] \longleftarrow M[1][col] \oplus H_\rho.duplex(M[1][col], b)$}end for
for ($row^0 \longleftarrow 3$ to $R - 1$) do  d Filling Loop: initializes remainder rows
    d Columns Loop: $M[row^0]$ is initialized; $M[row^1]$ is updated
    for ($col \longleftarrow 0$ to $C - 1$) do
        $rand \longleftarrow H_\rho.duplex(M[row^1][col] \boxplus M[prev^0][col] \boxplus M[prev^1][col], b)$
        $M[row^0][C-1-col] \longleftarrow M[prev^0][col] \oplus rand$
        $M[row^1][col] \longleftarrow M[row^1][col] \oplus$ rot($rand$)  d rot(): right rotation by $\omega$ bits
    end for
    $prev^0 \longleftarrow row^0$ ; $prev^1 \longleftarrow row^1$ ; $row^1 \longleftarrow (row^1 + stp)$ mod $wnd$
    if ($row^1 = 0$) then  d Window fully revisited
        d Doubles window and adjusts step
        $wnd \longleftarrow 2 \cdot wnd$ ; $stp \longleftarrow sqrt + gap$ ; $gap \longleftarrow -gap$
        if ($gap = -1$) then {$sqrt \longleftarrow 2 \cdot sqrt$}end if  d Doubles $sqrt$ every other iteration
    end if
end for
d Wandering phase: Iteratively overwrites pseudorandom cells of the memory matrix
    d Visitation Loop: $2R \cdot T$ rows revisited in pseudorandom fashion
for ($wCount \longleftarrow 0$ to $R \cdot T - 1$) do
    $row^0 \longleftarrow$ lsw($rand$) mod $R$ ; $row^1 \longleftarrow$ lsw(rot($rand$)) mod $R$  d Picks pseudorandom rows
    for ($col \longleftarrow 0$ to $C - 1$) do  d Columns Loop: updates $M[row^{0,1}]$
        d Picks pseudorandom columns
        $col^0 \longleftarrow$ lsw(rot$^2$($rand$)) mod $C$ ; $col^1 \longleftarrow$ lsw(rot$^3$($rand$)) mod $C$
        $rand \longleftarrow H_\rho.duplex(M[row^0][col] \boxplus M[row^1][col] \boxplus M[prev^0][col^0] \boxplus M[prev^1][col^1], b)$
        $M[row^0][col] \longleftarrow M[row^0][col] \oplus rand$  d Updates ftrst pseudorandom row
        $M[row^1][col] \longleftarrow M[row^1][col] \oplus$rot($rand$)  d Updates second pseudorandom row
    end for  d End of Columns Loop
    $prev^0 \longleftarrow row^0$ ; $prev^1 \longleftarrow row^1$  d Next iteration revisits most recently updated rows
end for  d End of Visitation Loop
d Wrap-up phase: output computation
$H.absorb(M[row^0][0])$  d Absorbs a ftnal column with full-round sponge
$K \longleftarrow H.squeeze(k)$  d Squeezes $k$ bits with full-round sponge
return $K$  d Provides $k$-long bitstring as output
```

By the user, allowing Lyra2's execution time to be fine-tuned according to the target platform's resources.

4. Application information in the UTXO trading model

4.1 Application Deployment Principles

Based on the UTXO trading model, the AICHAIN can expand a space after the transaction's output VOUT data. The space is used to store AI application description on the blockchain. That can allow a transaction to deploy one application to the blockchain with different version of transaction.

The information of the application unit needs to be included:

1、The type of application : provide services directly or provide executable files.

2、The address to access: provide the address of the service or the access address of the executable resource directly

3、the description of application: name、version、cost and other information.

4、Owner's public key: The unique address used to generate the application.

Only part of application's information will be stored on blockchain, in order to save the storage resource of blockchain.

The transaction ID is used as the position ID for an application unit.

Acquiring eligibility or rights to use one application by transferring AICHAIN's token (AIT) to the addresses of this application unit.

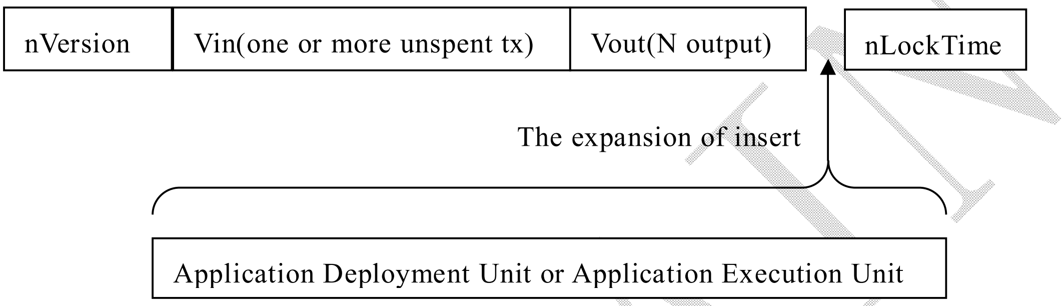
4.2 The expansion of transaction data

The transaction data based on UTXO model is mainly composed of two parts: input and output.

Input: consists of one or more unspent transaction information (unspent_tx_id, vout_n) and signature data.

Output: consists of one or more transfer destination addresses (script_pubkey), and the value of the AIT output to each address.

The transactional data in the AICHAIN has been expanded to add the area of APP units after VOUT outputs data, which can be used to deploy one APP unit.


nVersion
Vin(one or more unspent tx)
Vout(N output)
nLockTime
The expansion of insert
Application Deployment Unit or Application Execution Unit

The fifth bit of the nibble of nVersion is used as a flag to deploy the application unit. When this bit is set to 1, it indicates that the transaction contains at least one APP application deployment unit.

The condition is: (nVersion & 0x08000000)> 0 holds, indicating that the transaction contains the APP application deployment unit.

The sixth bit of the most significant byte of nVersion is used as the flag of the application execution unit. When this bit is 1, it indicates that the transaction contains at least one APP execution unit.

The conditions are: (nVersion & 0x04000000)> 0 holds, indicating that the transaction contains the APP application execution unit.

4.3 Application Deployment Unit : ADU

The APP application deployment unit contract name: ADU.

4.3.1 Customizable data content

AICHAIN allows third party applications to get data space on a block chain by publishing ADU applications, which is used to store [key and value] data. Third party applications can customize these data field names and content meanings to meet the needs of application system.

Field name	Type definition	Description
ADU_PUBKEY	std::vector<unsigned char> pubkey	public key of private key by ADU deployer
ADU_TYPE	uint32	ADU type: 0:reserved 1:data store Others: reserved
ADU_KEY_NAME[0]	std::vector<char> keyname keyname[0] is the length	Name of key
ADU_KEY_VALUE[0]	std::vector<unsigned char> keyvalue keyvalue[0] is the length	Value of key
…	…	
ADU_KEY_NAME[n]	std::vector<char> keyname keyname[0] is the length	Name of key
ADU_KEY_VALUE[n]	std::vector<unsigned char> keyvalue keyvalue[0] is the length	Value of key

- ADU_PUBKEY: The public key is generated by the application publisher using one's own private key. The first byte describes the length of the public key data followed by the public key data.

For example: public key hexadecimal data:

03cc149f66520680d85e18a01a8261a2746ee45fb5fb07ad13e9c316e1c955553d

The value of the first byte indicates the length of the entire public key data:

```
chHeader=pubkey_bin[0];
if (chHeader == 2 || chHeader == 3)
      return 33; // total length = 33 bytes
if (chHeader == 4 || chHeader == 6 || chHeader == 7)
return 65; // total length = 65 bytes
```

- ADU_TYPE：the type of application deployment unit. Currently this value is defined as 1. All other values are reserved.

- On the block chain, the above data is stored in the database by the following information fields:

ADU_ADDR	**ADU_PUBKEY**	**keyname**	**keyvalue**
ADU_ADDR_user	ADU_PUBKEY_user	ADU_KEY_NAME[0]	ADU_KEY_VALUE[0]
		…	…
ADU_ADDR_user	ADU_PUBKEY_user	ADU_KEY_NAME[n]	ADU_KEY_VALUE[n]

[key:value] is corresponding to a PUBKEY; each PUBKEY has different [key:value] data.

All data contents corresponding to each custom data unit ADU are identified by ADU_ADDR.

4.3.2 Application case: standard application unit information

For intelligent application resources, the following [key:value] fields can be defined in the ADU of the standard application unit information in accordance with the rules of the 4.3.1 section:

Keyname	KeyValue	Description
AIUNIT_TYPE	uint32	in section 5.4
AIUNIT_DATA_HASH	uint32	The HASH value of the application and data resource data
AIUNIT_FEE	int64	Fee of usage of resource. unit is 0.00000001 AIT
AIUNIT_NAME	std::vector<char> name name[0] is the length	AI application unit name
AIUNIT_INFO_URL	std::vector<unsigned char> info info[0] is the length	A url to get info in details.

- AIUNIT_TYPE: The standard application type is used to indicate that the standard application unit can be executable file, data resources, runtime platform and direct services(micro-SOA). It is currently defined as the 4 kinds of AI units.

- AIUNIT_HASH: The HASH value of the AI unit data is used to check the data of the application and prevent the tampering. When the application data changes, you need to issue an AEU to update this key’s value to the data area corresponding to the ADU unit.

- AIUNIT_FEE: The tariff is how many AI coins you need to pay when using this ADU, the unit is: 0.00000001 AIT (minimum unit).

- AIUNIT_INFO_URL：

application’s name, the introduction of function, the location url of application resources to obtain, and instructions for use, etc. This url is specified as JSON body returned for HTTP GET request with agreed format. Defined as follows:

```
{
    "version":"1.0.1",
    "name":"Dog or Cat?",
    "resource":"https://myapplication.com/DogCat.zip",
    "description":"This is an application to know it is a dog or cat from image",
    "runtime":[
        {
            "vm_type":"docker",
            "version":"0.01"
        }
    ]
}
```

Where resource is the address of the application resource.

4.4 Application execution unit : AEU

The agreed name of the application execution unit is AEU.

4.4.1 Custom data modification execution unit information

The user is allowed to change the content of the corresponding custom data area by sending a transaction to a specified ADU address.

The read operation do not need send a transaction, and can read directly.

Writing and deleting operations are required to send transactions before they can be executed.

- Reading operations, there is no real transaction, and users get data directly from the local node interface.

Field name	Type definition	Description
AEU_SIGNATURE	std::vector<unsigned char> signature signature[0] is the length	Signature of AEU data content
ADU_txid	uint256	Txid of ADU deploying on blockchain.
usrPubKey	std::vector<unsigned char> pubkey	The public key of user's private key when sending AEU
OP_CODE=1	uint8	0：reserved 1: read
ADU_PUBKEY_user	std::vector<unsigned char> pubkey pubkey[0] is the length	To read values of keyname belonging to this pubkey Can be null, then will read all keyname's value of all

		pubkeys.
ADU_KEY_NAME	std::vector<char> keyname keyname[0] is the length	Name of key Can be null, then will read all keyname’s value of the pubkey

This request corresponds to the read:

ADU_ADDR:usrPubKey:ADU_KEY_NAME:keyvalue

AEU_SIGNATURE: It is the result of the signature of the above AEU’s data area after the AEU_SIGNATURE field. Use usrPubKey for signature verification.

- The delete operation generates real transactions and modifies the database content of the block chain, It is only allowed to operate on the data in the executor's own public key range:

Field name	Type definition	Description
AEU_SIGNATURE	std::vector<unsigned char> signature signature[0]是长度	Signature of AEU data content
ADU_txid	uint256	Txid of ADU deploying on blockchain.
usrPubKey	std::vector<unsigned char> pubkey	The public key of user’s private key when sending AEU
OP_CODE=2	uint8	2: delete
ADU_KEY_NAME[0]	std::vector<char> keyname	name of key

	keyname[0] is the length	
…	…	
ADU_KEY_NAME[n]	std::vector<char> keyname keyname[0] is the length	Name of key

- Write operation need send a transaction and will modifie the database content of the block chain, allowing only the operation of the data under the executor's own public key:

Field name	Type definition	Description
AEU_SIGNATURE	std::vector<unsigned char> signature signature[0]是长度	Signature of AEU data content
ADU_txid	uint256	Txid of ADU deploying on blockchain.
usrPubKey	std::vector<unsigned char> pubkey	The public key of user’s private key when sending AEU
OP_CODE=3	uint8	3: write
ADU_KEY_NAME[0]	std::vector<char> keyname keyname[0] is the length	Name of key
ADU_KEY_VALUE[0]	std::vector<unsigned char>	Value of key

	key_value key_value[0] is the length	
…	…	
ADU_KEY_NAME[n]	std::vector<char> keyname keyname[0] is the length	Name of key
ADU_KEY_VALUE[n]	std::vector<unsigned char> key_value key_value[0] is the length	

4.4.2 Application case: standard application executive unit information

When a user want to use standard smart applications, need send a transaction with AEU, which does not need change ADU’s custom data area. The [keyname:keyvalue] corresponding to the write operation AEU unit is empty. There may be third party applications that require users to submit data when they need to be launched, which can be submitted through [keyname:keyvalue] in the AEU.

In addition, the usrPubKey in AEU will be used to distinguish user identity. When invoking the service provided by the application provider, the public key is required to verify the identity. Meanwhile, the public key and private key(usrPrivKey) can be used to encrypt data communication with the application provider.

5. Application deployment and implementation

5.1 ADU location identifier

When we need to use an app, we need to locate the application unit in the blockchain firstly. Based on UTXO model, when the application unit is deployed in the transaction data, the only one identifier is the transaction ID when deploying the application:

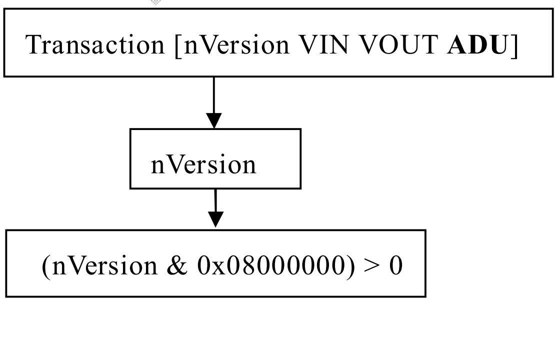
Field name	Type definition	Implication
ADU_txid	uint256	Transaction ID for deploying the application

When a user wants to obtain the corresponding application, he can use ADU_txid to find the corresponding transaction data in the blockchain, and then extract the ADU unit information from the corresponding transaction data.

5.2 P2SH address of ADU unit

When we need to use an application, you need to transfer the AIT to the address of the application. In order to make full use of blockchain transactions to record the usage history of application units, it is necessary to ensure the uniqueness of the corresponding address of the application. All transactions with this application address can be audited separately.

When deploying application, we need create a transaction with a specific nVersion as below, then we can put the ADU information unit into the transaction data.


Transaction [nVersion VIN VOUT **ADU**]
nVersion
(nVersion & 0x08000000) > 0

When creating the transaction, the ADU unit information contains a public key field: ADU_PUBKEY, which is provided by the deployer. The deployer holds the corresponding private key: ADU_PRIKEY.

After creating the original transaction, we can know the transaction ID:

Field name	Type definition	Implication
tx_hash	uint256	Transaction ID created when deploying the application

This transaction ID is the HASH of the transaction itself, this value participates in the generation operation of ADU address as a private key:

Tx transaction ID, 256 bits of data as private key: PrivKey_TX

The public key generated with this private key: PubKey_TX

Use this public key and ADU_PUBKEY to generate a P2SH-type address with a M/N (both M and N are 2) multiple signature: ADU_ADDR. This address corresponds to the address of the ADU application unit.

People who have the private key, ADU_PRIVKEY, can control the AIT assets on the ADU_ADDR. When making the transfer, they only need to use ADU_PRIVKEY and PrivKey_TX to perform multiple signature on the transfer transaction.

This definition can guarantee the uniqueness of each application address, and is bound to the location information of blockchain.

5.3 Operating environment of the application

The AICHAIN node is equipped with a docker operating environment and standard docker IMG as the basic operating platform. Operating environments of various programming languages are pre-built in the IMG, and the supporting programming languages and versions are released as a unified specification for public, including the application automatically downloaded after docker IMG launches, and the entrance and rules of implementation. Therefore, developers can develop and debug their own application in the docker environment, and the docker operating environment can be updated and continuously improved with the upgrade of the AICHAIN node.

At the same time, in order to the convenience of later extension and reserve more free customized space for application publishers, the application types of ADU include an entrance for providing the service directly, without operating environment for support.

5.4 Types of applications

5.4.1 Provide executable files

Specify the application type definition：ADU_TYPE=1

An executable package that is applicable to the docker IMG environment, which is typically in the application developer's perspective include 2 parts: application deployment + application execution.

The agreed rule is:

The executable package encapsulation format is zip

The agreed application type is: webapp, of which terminal using the browser to show the application.

Provide ADU_start.sh as a unified running entry in the zip wrapper inner root directory

In this way, the application developer can customize the ADU_start.sh script in the executable package for the deployment and startup of the application, based on the docker IMG environment of the AICHAIN node. There is plenty of flexibility and customization.

5.4.2 Provide direct access to services

Specify this application type definition: ADU_TYPE = 2

The direct service entry is: the web app mode, which is displayed in the browser

The application does not need to download the executable file data, and the url address of resource in ADU's detailed information is a direct access to the service.

5.4.3 Provides running platform resources

Specify this application type definition: ADU_TYPE = 3

Allows the service providers that have personalized and customized application running environment resources to publish their own application running platform resources to the blockchain.

When an agreed user makes a transaction using this resource, the provider needs to assign the user an application running platform that allows the user to enter the “resource” address for the application resource to run on the running platform.

5.4.4 Data resources

Resource conventions for data classes ADU_TYPE = 4

Allow vendors with tagged data resources to publish data resource descriptions to the blockchain for those who need to use data for deep learning.

When an agreed user makes a transaction using this resource, the provider needs to provide the user with a data resource download address (resource address) that allows users to download data resources. The provider can also verify the user identity of the data resource for this download class.

5.4.5 Extend other types of resources

The current maximum value of ADU_TYPE is not more than 0x0000ffff, and the first two bytes are reserved for later use.

We agree that other resource types expand after this, starting from ADU_TYPE = 5 to 0x0000ffff.

5.5 Using the ADU application

5.5.1 Transaction data containing AEU units

When you need to use an ADU application, you need to transfer agreed amount of AIT to this ADU_ADDR address.And to contain information about the AEU unit in this transaction, Use the following ADU_tx_id information to specify the ADU application to use.

This is a standard transfer transaction data, in order to identify the transaction contains AEU unit, We agree that the sixth bit of the highest byte of "nVersion" is the symbol, when the bit to 1, said The transaction contains AEU unit.

For the above special deals using APP applications, the nodes of the AICHAIN are received, firstly, using ADU_tx_id to extract the transaction data from the APP application deployment, verify that the address output to the APP in the transfer transaction is correct (according to the rules of chapter 5.2).Then the APP application unit information is extracted to verify whether the amount of AIT exported to the APP application address meets the agreed cost in the ADU application unit in the transfer transaction.

Through the above verification check, the nodes of the AICHAIN will perform two actions: transferring the transfer transaction to the network and running the APP application.(It's ok for users to run the APP later).

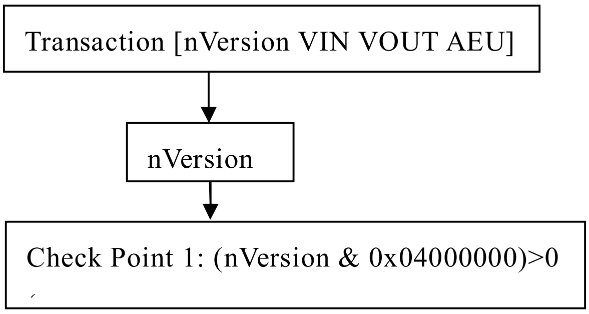
5.5.2 Transaction data verification

When you want to use an ADU application, you need to create a transaction to transfer a specific amount of AIT to this ADU_ADDR address, including the AEC Execution Unit.

There are some check points for this transaction data:

1. Transaction version number check

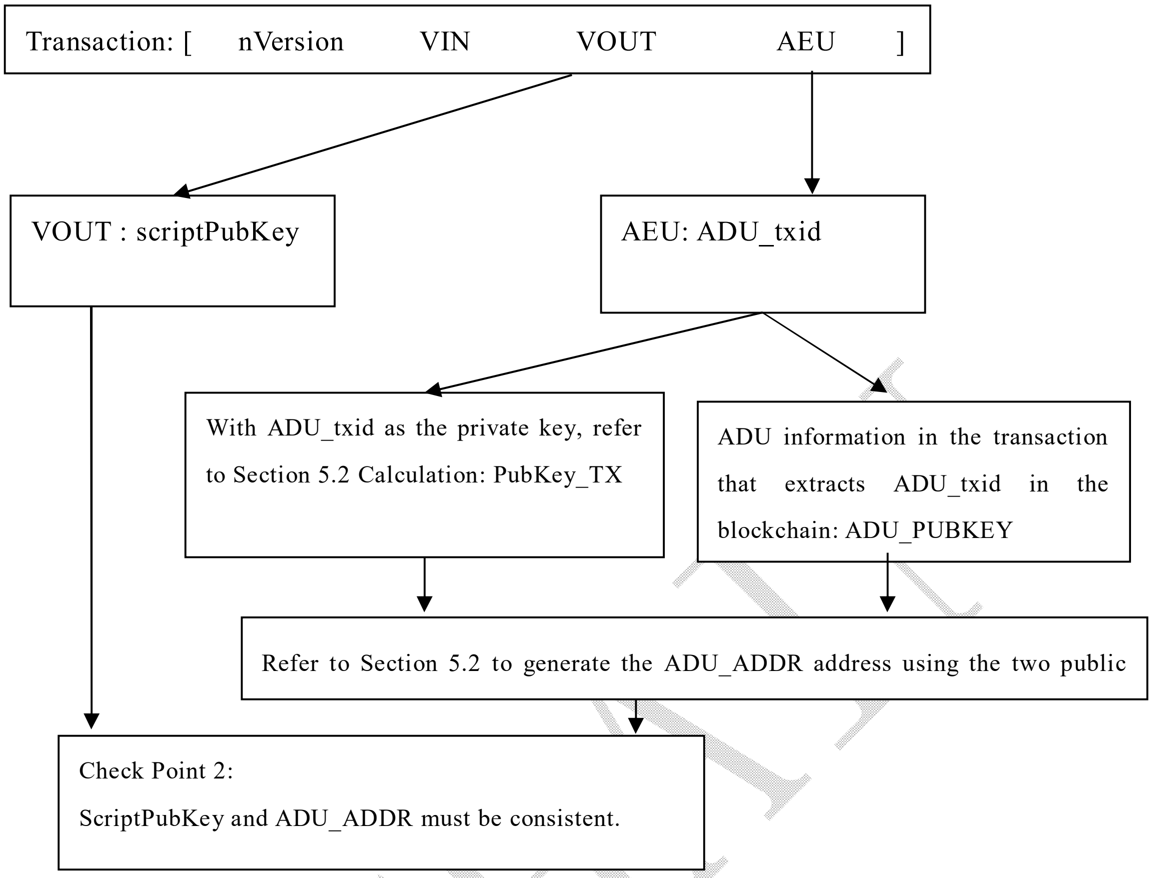
The conditions are: (nVersion & 0x04000000)> 0 must be true.


Transaction [nVersion VIN VOUT AEU]
nVersion
Check Point 1: (nVersion & 0x04000000)>0

2. The output address of the transaction is same as the address of the ADU specified in the AEC execution unit:

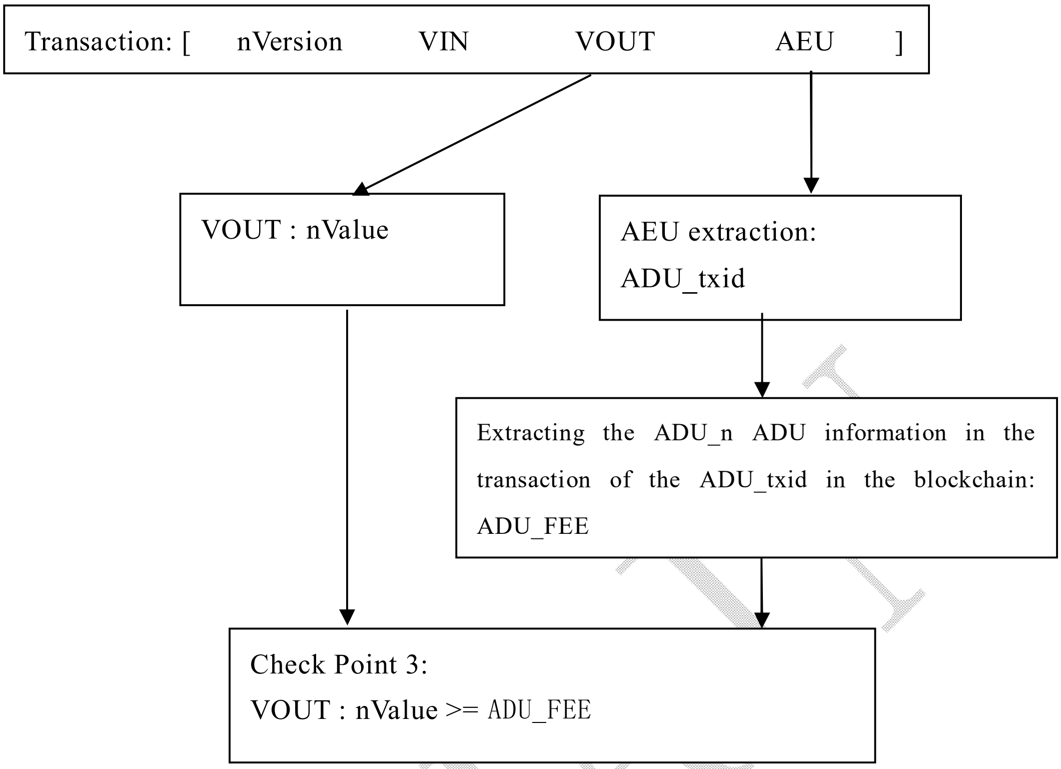
By the use of ADU_txid in AEC execution unit, we can extract ADU data from the block chain. We can refer to section 5.2 ADU_ADDR Address, compared with the output address of the vout, which must be same.

When the transaction has multiple VOUT cells, it needs to chek the output address of each VOUT. If one output address matches the address of ADU_ADDR, this means it is right for ADU, and the amount of AIT for transfer to this VOUT is recorded.


Transaction: [nVersion VIN VOUT AEU]
VOUT : scriptPubKey
AEU: ADU_txid
With ADU_txid as the private key, refer to Section 5.2 Calculation: PubKey_TX
ADU information in the transaction that extracts ADU_txid in the blockchain: ADU_PUBKEY
Refer to Section 5.2 to generate the ADU_ADDR address using the two public
Check Point 2:
ScriptPubKey and ADU_ADDR must be consistent.

3. The amount of AI coins that ADU address received in this transaction is sufficient:

On the basis of the second step, check the amount of AI coins received at the ADU_ADDR address in the vout, which must be greater than or equal to the ADU_FEE value in the ADU unit information.


Transaction: [nVersion VIN VOUT AEU]
VOUT : nValue
AEU extraction:
ADU_txid
Extracting the ADU_n ADU information in the transaction of the ADU_txid in the blockchain: ADU_FEE
Check Point 3:
VOUT : nValue >= ADU_FEE

On the basis of the transaction signature verification, the above three check points must be satisfied at the same time to be judged as legal transaction. Otherwise, the AICHAIN node will reject the transaction and ensure the consistency of the AEC execution information and the transaction vout information.

5.5.3 The application provider verifies the identity of the user

When you need to use the application, we can provide the application service a means to verify the user identity information. Use the ADU_PUBKEY in the ADU unit and the user's own private key: USER_PRIVKEY, and use the ECC asymmetric encryption algorithm to decrypt the random password issued by the application service provider. After entering the correct random password, you can begin to use the application service.

Convention: When a user want to use an application, he must pass his own public key while accessing the webapp portal as a parameter.

Convention: The public key provided by the user must be the public key corresponding to the signature private key used when creating the transaction of sending AIT to the ADU_ADDR. Such application providers have the ability to verify AIT payment information.

For example:

https://myapplication.com/index.html?usrPubKey=03cc149f66520680d85e18a01a8261a2746ee45fb5fb07ad13e9c316e1c955553d

Parameter name	Type	meaning
usrPubKey	Hexadecimal string	the public key data corresponding to the user's private key

The application server uses the above information to generate a random access password that is passed to the user after encryption, invoked by the interface to decrypt, and or after decrypting by use of an offline decryption tool, submit to enter the normal application for use.

Application Encryption: Use USER_PUBKEY and ADU_PRIVKEY

Consumer Decryption: Use ADU_PUBKEY and USER_PRIVKEY

Application providers can customize their own authentication system as they want.

5.6 Running process

The AICHAIN has four roles: data provider, application provider, running platform resource provider, resource consumers. The first three are different types of resources. Resource consumers are users who use these ADU resources. Consumers can be ordinary individuals or companies that are developing AI applications(A lot of data is needed for machine learning)

Define the following 4 companies (or individuals):

A: It is a company specialized in image tagging, which expects that make money from the data needed by AI developing.

B: It is a company that specializes in developing AI applications, which expects to rely on the developed AI applications to make money.

C: It is a company that holds a lot of server with graphics cards and running environment of tensorflow and caffe platform. It expects to rely on these resources of running platform to make money.

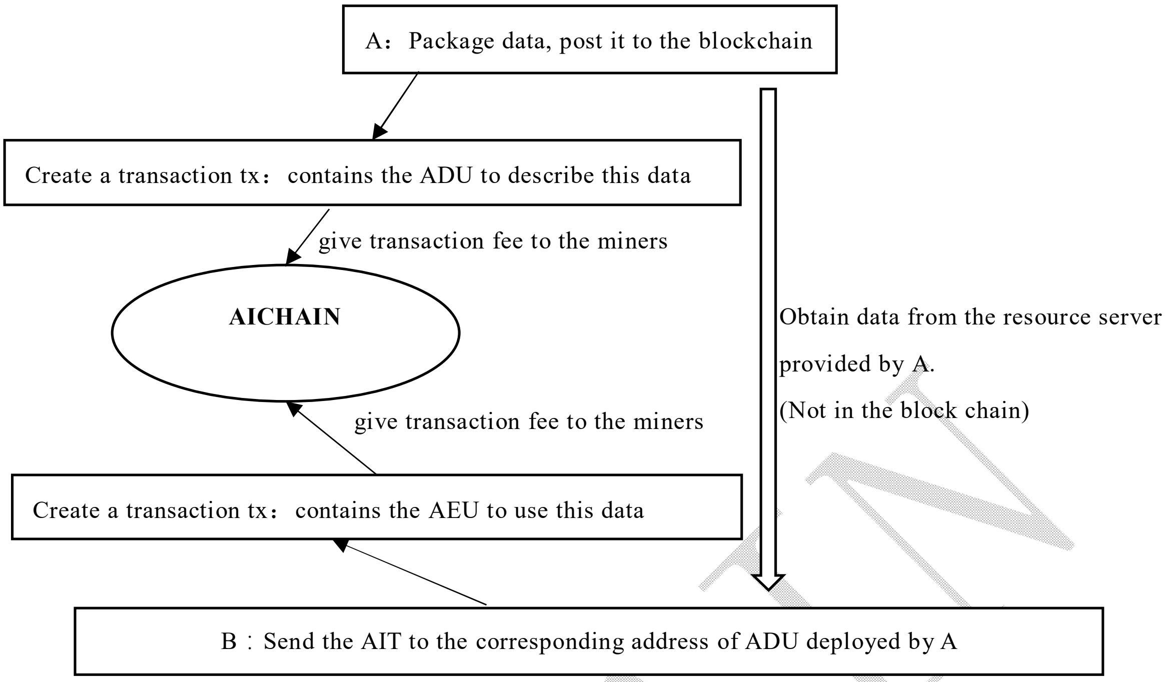
D: He is a normal user, has a bunch of pictures of cats and dogs in his hand and hopes to find a tool to help him classified storage.

5.6.1 Developers use data resources

Developers need data to complete machine learning and develop AI applications.

A: data provider

B: resource consumer


A：Package data, post it to the blockchain
Create a transaction tx：contains the ADU to describe this data
give transaction fee to the miners
AICHAIN
give transaction fee to the miners
Create a transaction tx：contains the AEU to use this data
Obtain data from the resource server provided by A.
(Not in the block chain)
B：Send the AIT to the corresponding address of ADU deployed by A

A: have the picture of dogs and cats, and the mark result of each image. (the mark is about whether it's dog or cat in the picture)

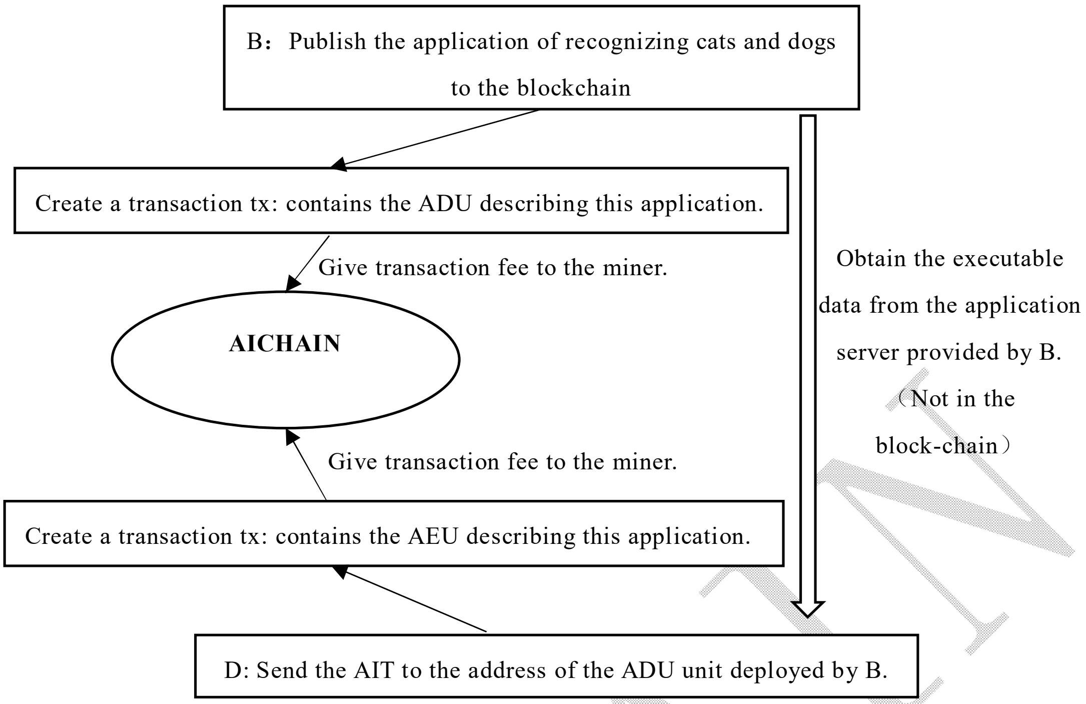
B: the developer of this AI application that design to identify cats and dogs from pictures, B is a company.

5.6.2 Applications for users.

This application is for users to use the AI application of recognizing cats and dogs on their own computers.

B: the application provider.

D: the user.


B：Publish the application of recognizing cats and dogs to the blockchain
Create a transaction tx: contains the ADU describing this application.
Give transaction fee to the miner.
AICHAIN
Give transaction fee to the miner.
Create a transaction tx: contains the AEU describing this application.
D: Send the AIT to the address of the ADU unit deployed by B.
Obtain the executable data from the application server provided by B.（Not in the block-chain）

B: Deploy the information of the completed application of recognizing cats and dogs to the block-chain, and provide AI application executable file data for download.

D: Because the user has a computer with a graphics card, it can support running the AI application of recognizing cats and dogs.

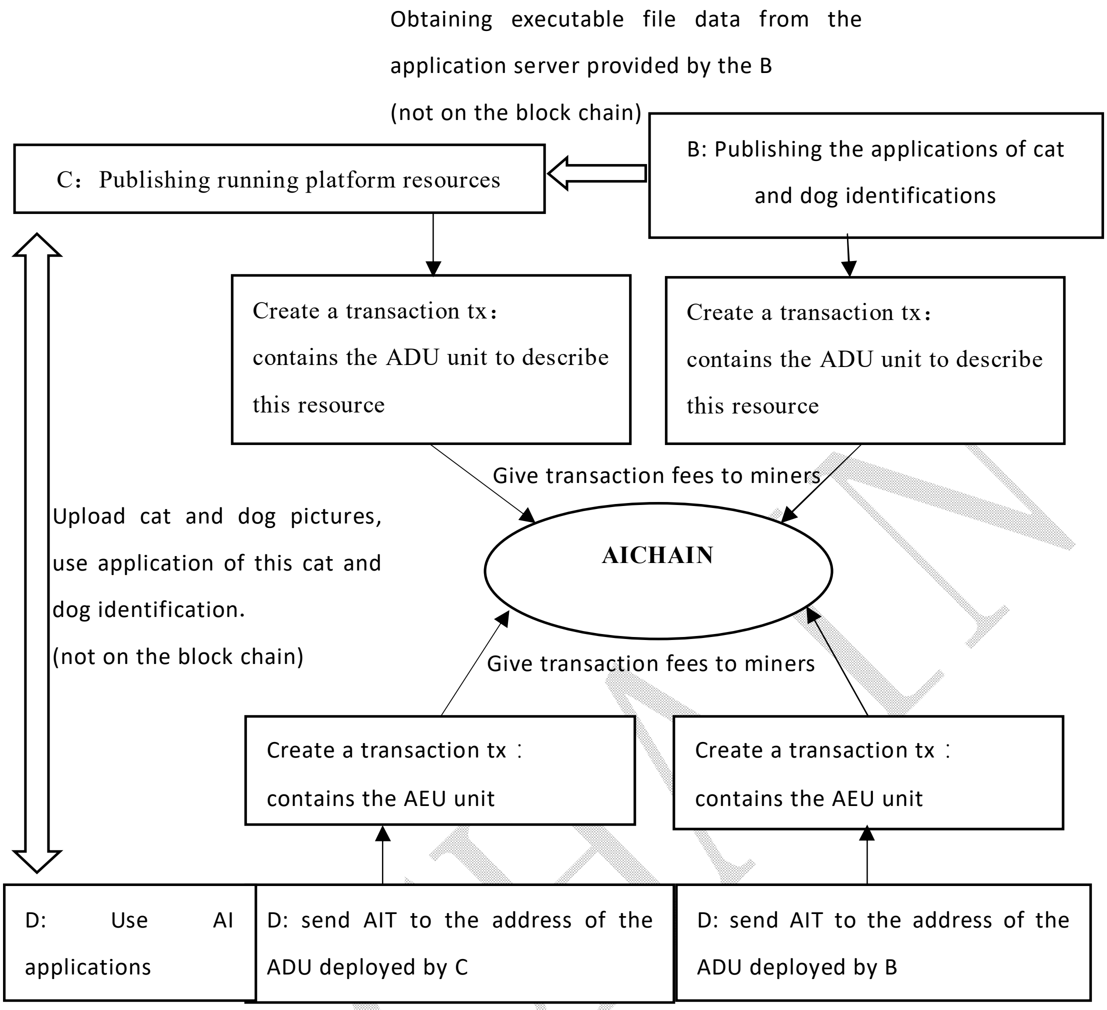
5.6.3 users use the resources and applications of running platforms

AI application of cat and dog image recognition on the running platform of the third party.

B：is an application provider

C：is the resource provider of the running platform

D：users


Obtaining executable file data from the application server provided by the B (not on the block chain)
C：Publishing running platform resources
B: Publishing the applications of cat and dog identifications
Create a transaction tx：contains the ADU unit to describe this resource
Create a transaction tx：contains the ADU unit to describe this resource
Give transaction fees to miners
Upload cat and dog pictures, use application of this cat and dog identification. (not on the block chain)
AICHAIN
Give transaction fees to miners
Create a transaction tx：contains the AEU unit
Create a transaction tx：contains the AEU unit
D: Use AI applications
D: send AIT to the address of the ADU deployed by C
D: send AIT to the address of the ADU deployed by B

B: Deploy the cat and dog identification (AI application information)to the block chain and provide the downloading of data in the AI application executable file .

D: the user does not have the right computer to run AI applications of the cat dog identification. You need to borrow the platform provided by C. You need to transfer to the address of the running resource ADU provided by the C.

C: allocate resources on a running platform of AI applications for a video card server to D.

D: Transfer AIT to the address of the ADU unit of the cat dog identification application.

D: D initiates and runs the AI applications of cat dog identification provided by B on the AI application running platform assigned by the C. Then you can upload pictures to complete the classification.

Reference

[1] http://www.coinwarz.com/cryptocurrency/

[2]https://github.com/leocalm/Lyra/blob/master/Lyra2/Lyra2ReferenceGuide.pdf

[3] https://bitcointalk.org/index.php?topic=586407.0

[4] http://phoenixcoin.org/archive/neoscrypt_v1.pdf

[5] https://en.bitcoin.it/wiki/Category:History

[6] https://github.com/bitcoinbook/bitcoinbook

[7] https://en.bitcoin.it/wiki/Wallet_import_format

[8] https://en.bitcoin.it/wiki/List_of_address_prefixes

[9] https://en.bitcoin.it/wiki/Transaction

[10] https://github.com/ethereum/wiki/wiki/White-Paper

[11] https://en.bitcoin.it/wiki/Pay_to_script_hash

[12] https://en.bitcoin.it/wiki/Transaction#Pay-to-PubkeyHash

[13] https://github.com/bitcoin/bips/blob/master/bip-0013.mediawiki

[14] https://en.bitcoin.it/wiki/Multisignature

[15] https://docs.docker.com/engine/userguide/eng-image/baseimages/

[16]https://docs.docker.com/engine/userguide/eng-image/multistage-build/

[17] https://gist.github.com/ericjang/959c03168c0bdfac1ca3

[18]https://github.com/tensorflow/tensorflow/blob/master/tensorflow/tools/docker/README.md

[19]https://github.com/tensorflow/tensorflow/tree/master/tensorflow/tools/docker

[20] https://hub.docker.com/r/bvlc/caffe/

[21] https://github.com/BVLC/caffe/tree/master/docker

[22]http://tleyden.github.io/blog/2014/10/25/running-caffe-on-aws-gpu-instance-via-docker/

[23] https://en.wikipedia.org/wiki/Elliptic-curve_cryptography