**Using** [**Abel-Ruffini theorem**](https://en.wikipedia.org/wiki/Abel%E2%80%93Ruffini_theorem) **on encryptions**

**Introduction**

Let’s consider the following polynomial equation:

Where , , , , and are rational numbers.

It has the following roots(following the same ordering): , , , , , , , ,

The fully expanded form of the polynomial equation will be this:

Using [Vieta’s formulas](https://en.wikipedia.org/wiki/Vieta%27s_formulas):

And the rest of the coefficients can be calculated in the same manner.

As [Abel-Ruffini theorem](https://en.wikipedia.org/wiki/Abel%E2%80%93Ruffini_theorem) states that there’s no solutions in radicals to general polynomial equations of degree 5 or higher with arbitrary coefficients, **there’s no direct formula for finding all the roots of such equations analytically if** [**Bring radical**](https://en.wikipedia.org/wiki/Bring_radical)**’s not allowed**, meaning that **such roots can act as raw passwords, while their resulting polynomial equation can behave like the encrypted version**.

Therefore, a set of roots as a password can be correct only if those roots can produce the expected polynomial equation.

More formally speaking, let , where is the set of [constructible](https://en.wikipedia.org/wiki/Constructible_number) polynomial equations of degree n, where , be an encryption function that converts n [constructible](https://en.wikipedia.org/wiki/Constructible_number) roots(where complex numbers are allowed) into a polynomial equation of degree n.

Then by [Abel-Ruffini theorem](https://en.wikipedia.org/wiki/Abel%E2%80%93Ruffini_theorem), there doesn’t exist , where , such that it can be the corresponding decryption function, meaning that can be a [one-way function](https://en.wikipedia.org/wiki/One-way_function) when n is very large.

**Note that, because the resulting polynomial is stored as a special data type, where its equality must be *exact literal expression matches* rather than just mathematical identities, *the ordering among all distinct roots can also matter*.**

For instance, if the ordering of the roots on the above example is reversed, then applying [Vieta’s formulas](https://en.wikipedia.org/wiki/Vieta%27s_formulas) again will yield these results instead:

, which doesn’t literally match the previous expression , even though they’re mathematically identical.

Similarly:

, which doesn’t literally match the previous expression either, even though they’re also mathematically identical.

The same applies to the rest of the coefficients.

**For the same reason, *the literal expression of a root can matter as well*.**

For instance, even though and are mathematically identical, they’re still 2 different literal expressions leading to polynomial equations not literally matching each other.

To further illustrate this, roots , , , are mathematically identical to , , , , but the differences in their literal expressions will lead to polynomial equations not literally matching each other, as can be seen by applying [Vieta’s formulas](https://en.wikipedia.org/wiki/Vieta%27s_formulas) with these roots:

These 2 expressions don’t literally match each other even though they’re mathematically identical too.

On the other hand, **the ordering among each repeated root doesn’t matter**, at least in most cases, because they’ll always form the same parts of the literal expression in the resultant polynomial equation.

**Therefore, *repeated roots should be avoided as much as possible*, otherwise there will be** [**collisions**](https://en.wikipedia.org/wiki/Collision_(computer_science)) **when it comes to the right ordering among all roots.**

**Decryption Difficulties**

**Decrypting the resultant polynomial equation as the encrypted form means finding *all its roots* with the *right ordering among all distinct roots* and *all the right root literal expressions*.**

It’s because, as shown in the introduction, just finding all its roots can only lead to the mathematically identical polynomial equation, but possibly without literal expression matches.

Only when the root ordering and all the root literal expressions are also right, the resultant polynomial equation expression can literally match the expected one, at least in a general sense.

This means there are 3 obstacles in decryption:

1. Finding all the roots of a given polynomial equation, which is known to have no direct formula to do so analytically when its degree is 5 or higher due to [Abel-Ruffini theorem](https://en.wikipedia.org/wiki/Abel%E2%80%93Ruffini_theorem), and **factoring a polynomial of degree n with named constants into n factors analytically is generally significantly slower than the reverse process**
2. Finding the right ordering among all distinct roots, which demands [factorial time](https://en.wikipedia.org/wiki/Time_complexity#Factorial_time) if the brute force approach’s used, assuming that there’s no more efficient algorithms on determining such ordering yet
3. Finding the right literal expressions of all the roots, which can demand up to [exponential time](https://en.wikipedia.org/wiki/Time_complexity#Exponential_time) if the brute force approach’s used, assuming that there’s no more efficient algorithms on determining such literal expressions yet

Bear in mind that the time complexity for factoring a polynomial of degree n with named constants into n factors analytically ***demands further researches***.

**Of course, it might be possible for** [**collisions**](https://en.wikipedia.org/wiki/Collision_(computer_science)) **to occur in 2. or 3., or among 2. and 3., and such potential** [**collision**](https://en.wikipedia.org/wiki/Collision_(computer_science)) **issues *demand further researches*, including but not limited to their *triggering conditions*, *the number of collisions under certain circumstances*, etc.**

**Also, whether there can be algorithms on finding the right ordering among all the roots and/or the right expressions of all the roots that are much, much faster than the brute force approaches *demands further researches*.**

On the other hand, it’s obviously **impossible to have collisions on 1.**, because it’s impossible for different sets of roots to form the mathematically identical polynomial equation, which is just a corollary of the [fundamental theorem of algebra](https://en.wikipedia.org/wiki/Fundamental_theorem_of_algebra#Corollaries).

For those who try to use [numerical methods](https://en.wikipedia.org/wiki/Root-finding_algorithms) to find all the roots of a given polynomial equation, those solutions are extremely unlikely to produce the polynomial equation with its expression literally matching the expected one, because they’re meant to be approximations rather than being absolutely precise, therefore **using numerical methods for decryption will almost always fail**.

As for applying the concept of [rainbow tables](https://en.wikipedia.org/wiki/Rainbow_table) to solve the resultant polynomial equation, it means pre-computing a set of roots with the exact ordering and literal expressions mapping to its resultant polynomial equation with the exact literal expression, for all possible polynomial equations that can be formed with a practical amount of time.

**But as there’s an infinite number of polynomial equations of a given degree in theory and many different literal expressions of the same polynomial equation, *it’s impractical to have an exhaustive list*.**

**It’s very hard to form a list of common sets of roots either, because of *the difficulty on defining what makes a set of roots common*, especially with the ordering among the distinct roots and the right literal expressions of all roots taken into account, so this reason also causes** [**dictionary attacks**](https://en.wikipedia.org/wiki/Dictionary_attack) **to become atrocious.**

**Encryption Efficiencies**

**Encrypting the raw root set means producing the resultant polynomial equation while *preserving the ordering among all the distinct roots and the literal expressions of all roots*.**

As shown in the decryption section, demanding literal expression matches, not just mathematical identities between the resultant polynomial equation and the expected counterpart, is to **drastically increase the time complexity involved in decryption**.

On the other hand, the encryption time complexity is determined by that of the [Vieta’s formulas](https://en.wikipedia.org/wiki/Vieta%27s_formulas), which produces all the coefficients of the resultant polynomial equation.

As shown in [this article](https://www.geeksforgeeks.org/vietas-formulas/), the encryption time complexity is , which is [quadratic time](https://en.wikipedia.org/wiki/Time_complexity#Polynomial_time).

**While such a time complexity is far from ideal on the encryption side, it’s still *trivial compared to those involved in the decryption counterparts*, which can be as high as** [**factorial time**](https://en.wikipedia.org/wiki/Time_complexity#Factorial_time)**, assuming that it’s indeed possible to find all the roots.**

Let’s say that , under quadratic time complexity, its growth is 400, while the same under is 1048576(assuming that every root has 2 different literal expressions), and the same under is 2432902008176640000.

**Password Check Efficiencies**

**Checking if the password’s correct is to determine *whether the expression of the resultant polynomial equation literally matches the expected one*.**

**Needless to say, *only mathematically identical polynomial equations can have literally matching expressions*.**

However, determining whether 2 polynomial equations has literally matching expressions are usually way, way easier than if they’re just mathematically identical, because **the former only demands literal comparisons**, while the latter demands mathematical comparisons, which is of course much, much more complicated and convoluted.

The literal comparison time complexity is still , which is again [quadratic time](https://en.wikipedia.org/wiki/Time_complexity#Polynomial_time), because the literal expressions of the resulting polynomial equation can be completely produced by just using [Vieta’s formulas](https://en.wikipedia.org/wiki/Vieta%27s_formulas), which is of the same time complexity, and comparing each atomic part of 2 literal expressions is of [constant time](https://en.wikipedia.org/wiki/Time_complexity#Constant_time).

**Therefore, the password check efficiencies are of the same magnitude as that of encryption efficiencies, meaning that *the time complexity of the former is still trivial compared to that of decryption*.**

**Encryption Strength From The Roots**

**First, *using all numeric literals as roots will lose all encryption strengths from different literal expressions among the same resultant polynomial equation*, because all the coefficients of such polynomial equations will always be the same numeric literals, which have only 1 possible literal expression.**

Similarly, as mentioned in the introduction section, **the encryption strength drops as the number of repeated roots and their** [**multiplicities**](https://en.wikipedia.org/wiki/Multiplicity_(mathematics)#Multiplicity_of_a_root_of_a_polynomial) **increase**, because such increases will lose more and more encryption strengths from the ordering among all distinct roots.

On the contrary, using only named constants as roots might risk exposing at least some, and in some really extreme cases, all, roots of the resultant polynomial equations, and the possibility of exposing the ordering among the distinct roots and even the right literal expressions of all roots can’t be completely negated, even though all these possibilities ***demand further researches***.

For roots having 2 literal expressions instead of 1, at least some of them probably come in [conjugate pairs](https://en.wikipedia.org/wiki/Conjugate_(square_roots)) because it’s unlikely that more complex constants will be used(like [casus irreducibilis](https://en.wikipedia.org/wiki/Casus_irreducibilis)), so while such roots will drastically increase the time complexity on finding the right literal expressions of all the roots, they also make finding the rest of the roots easier if some of them are already found.

Of course, this can be changed by mixing numeric literals with named constants, allowing irrational or even complex numbers to be coefficients of the resulting polynomial equations, but this can cause the password to become too complex to remember and/or use in practice, even though polynomial equations with complex coefficients can further increase the encryption strength.

**The strategies on the proportion between numeric literals and named constants, as well as what combinations of literals and constants will give higher encryption strength, *demand further researches*.**

The degree of the resultant polynomial equation also matters, and **the higher the degree, the higher the encryption strength**, at least in a general sense.

As there are [septic equations](https://en.wikipedia.org/wiki/Septic_equation#Solvable_septics) of certain types that are known to be solvable, the degree of the resultant polynomial equation should be at least 8, and **it should be as high as the password encryption and comparison can still be efficient enough**.

In other words, n should be as large as still isn’t too large, because both password encryption and comparison are of [quadratic time](https://en.wikipedia.org/wiki/Time_complexity#Polynomial_time).

**Converting Plaintext Passwords Into Roots**

**To take advantage of this password encryption scheme while letting password owners to use passwords in plaintext, *such passwords can be broken down into a list of characters, each being a named constant or a numeric literal corresponding to a root of the resulting polynomial equation, perhaps with some special characters and situational rules*.**

For instance, consider this password: 4(-6ci)d3(\_f+9)dd

It can be readily converted to be the following roots: 4, , , 3, , ,

Which leads to this polynomial equation:

While the concrete conversion scheme should be established on a case-by-case basis, the fact that **the length of the password in plaintext directly dictates the degree of the resulting polynomial equation** remains universally applicable.

**However, if there are special characters and/or situational rules, *no 2 passwords in plaintexts can lead to the same set of roots*, otherwise it should be assumed that** [**collisions**](https://en.wikipedia.org/wiki/Collision_(computer_science)) **will occur for such passwords.**

Also, because repeated roots lead to collisions among all possible ordering of all roots, **this password conversion scheme, without further amendment, will lead to** [**collisions**](https://en.wikipedia.org/wiki/Collision_(computer_science)) **of passwords in plaintexts whenever such passwords have duplicate characters**.

To solve this, if a password in plaintext has n characters, after breaking them into a list of characters, **at least some of the characters should be attached with an existing numeric literal or existing named, in the sense that *there will be no repeated roots***.

For instance, consider the same password: 4(-6ci)d3(\_f+9)dd

That password can be readily converted to be the following distinct roots after attaching some existing numeric literals and/or existing named constants:

, , , , , ,

Which leads to this polynomial equation:

At least the following should be noted:

1. What root should be attached by existing numeric literals and/or existing named constants ***demands further researches***
2. **Many different arithmetic operators should be used in such attachments**, but the ideal strategies on this ***demand further researches***

**The time complexity of this password conversion scheme, along with the root** [**multiplicity**](https://en.wikipedia.org/wiki/Multiplicity_(mathematics)#Multiplicity_of_a_root_of_a_polynomial) **check, should be controlled to** [**quadratic time**](https://en.wikipedia.org/wiki/Time_complexity#Polynomial_time)**, which is the same as the rest of the encryption steps, otherwise the whole encryption scheme would become too inefficient.**

However, with such existing numeric literals and/or existing named constants attached, there’s a new problem:

1. On one hand, **the way they’re attached to the list of characters to form roots should be random per password setup/change**, in order to maximize encryption strength
2. On the other hand, when the same password is expected, **the aforementioned way they’re attached must be exactly the same as that when setting up that expected password or changing the password to the currently expected one**, otherwise it’d be theoretically impossible to form the expected polynomial equation even with the right raw password input, because **recovering the way those existing** **numeric literals and/or existing named constants are attached is of** [**factorial time**](https://en.wikipedia.org/wiki/Time_complexity#Factorial_time)

Combining, this means **a digital certificate, which stores the way they’re attached for the current password of the given user ID, must be generated**, so losing that digital certificate means theoretical impossibility of login success, even if the client inputted the right password.

Upon setting up a password or changing it to a new one, **the digital certificate will be stored in the server, so the client terminal will request the server to send that digital certificate with the given user ID upon login attempts**.

Of course, this also means that, **once the attacker knows the ID of a user, that attacker can get the digital certificate of that user, thus knowing the way those existing numeric literals and/or existing named constants are attached and hence removing quite some decryption difficulties, unless the digital certificate is itself protected by a 2-way encryption**.

**However, the attacker still has to find all the roots with the right ordering and their right literal expressions, and *just knowing the aforementioned information is far from solving all the remaining obstacles*.**

Loosely speaking, **these attachments of numeric literals and existing named constants are like** [**salts**](https://en.wikipedia.org/wiki/Salt_(cryptography)) **and** [**peppers**](https://en.wikipedia.org/wiki/Pepper_(cryptography)) **in** [**hashing**](https://en.wikipedia.org/wiki/Cryptographic_hash_function) **passwords.**

In the case of the digital certificates that are stored in the server, they behave like [salts](https://en.wikipedia.org/wiki/Salt_(cryptography)); **The** [**pepper**](https://en.wikipedia.org/wiki/Pepper_(cryptography)) **counterpart will be a universal digital certificate that’s hardcoded into the client software source codes in a highly obfuscated manner.**

Because the [peppers](https://en.wikipedia.org/wiki/Pepper_(cryptography))are fixed while the [salts](https://en.wikipedia.org/wiki/Salt_(cryptography)) are random per password setup/change, [**peppers**](https://en.wikipedia.org/wiki/Pepper_(cryptography)) **should be applied first**, so the generated [salts](https://en.wikipedia.org/wiki/Salt_(cryptography)) can be controlled to ensure there won’t be repeated roots.

**Integrating With Traditional** [**Hash Functions**](https://en.wikipedia.org/wiki/Hash_function)

To make the conversion scheme even more secure, the password in plaintext can first be [hashed](https://en.wikipedia.org/wiki/Cryptographic_hash_function) with a [salt](https://en.wikipedia.org/wiki/Salt_(cryptography)) and a [pepper](https://en.wikipedia.org/wiki/Pepper_(cryptography)), then the hash can be further broken down into a list of characters, each having a unique reserved special character attached with an operator, before being converted into roots.

For instance, consider the same password: 4(-6ci)d3(\_f+9)dd

And a random salt generated upon this password setup/change(then stored in the server): 2YxwvUt5R9

**The highly secret and obfuscated algorithm(with its performance intentionally diminished with very powerful machines to make brute forcing even more inefficient) of combining the password in plaintext, the** [**salt**](https://en.wikipedia.org/wiki/Salt_(cryptography)) **in plaintext, and the** [**pepper**](https://en.wikipedia.org/wiki/Pepper_(cryptography))**(embedded in the client software source codes in a highly obfuscated manner) should be on the client software and lead to very, very complicated and convoluted results**, but for the sake of demonstration, let’s just assume that, without the use of a [pepper](https://en.wikipedia.org/wiki/Pepper_(cryptography)), it appends the latter and its ASC II code version at the end of the former, then appends the reverse of the former again at the end of the latter.

In this case, it should be this: 4(-6ci)d3(\_f+9)dd2YxwvUt5R9508912011911885116538257dd)9+f\_(3d)ic6-(4

It becomes the following after applying [SHA3-512](https://en.wikipedia.org/wiki/SHA-3): 0481ce93c4a19728ea2cb284e5c8e28f8be7c6c4edd492f1ad5e62fb365013edd20333a879c36e5a3fe2607dcb4f505c092f4b890d022fcd49a99d63bc67cd9d

Note that there are 10 numeric literals, 0-9, and 6 named constants, a-f, which are both far from being enough, so the hexadecimal number generated by [SHA3-512](https://en.wikipedia.org/wiki/SHA-3) should be converted into a base 256 number first, like the following:

to : Numeric literals 1 to 100

to : Named constants to

to : Named constants to

to : Named constants to

to : Named constants to

to : Named constants to

to : Named constants to

So 0481ce93c4a19728ea2cb284e5c8e28f8be7c6c4edd492f1ad5e62fb365013edd20333a879c36e5a3fe2607dcb4f505c092f4b890d022fcd49a99d63bc67cd9d will be converted into a base 256 number with 64 digits, which forms the following 64 roots:

04 to to 4, 81 to , ce to to , 93 to to , c4 to to , a1 to to , 97 to to , 28 to to 40, ea to to , 2c to to 44, b2 to to , 84 to to , e5 to to , c8 to to , etc…

The original hexadecimal number with 128 digits produced by [SHA3-512](https://en.wikipedia.org/wiki/SHA-3) can be further lengthened to have 1024 digits or more, like repeating it 8 times, in order to produce a base 256 number with 512 digits to further strengthen the encryption.

In the case of 512, there will be 512 literals, including 100 different numeric literals and 156 different named constants, to be combined with existing numeric literals and existing named constants via different ways and combinations of additions, subtractions, multiplications and divisions, to form 512 distinct roots, which should produce a resultant polynomial equation that is **very, very hard to factorize efficiently**.

**With this, regardless of the length of the original password in plaintext, the growth of the time complexity of the whole password encryption and comparison process will be always of , which is *262144*, while that of the whole password decryption process, assuming that it’s even possible in theory, will be of , which should be *big enough to make decryption virtually impossible in practice*.**

**Integrating With** [**RSA**](https://en.wikipedia.org/wiki/RSA_(cryptosystem))**/**[**AES**](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard)

To further reduce the chance of exposing the resultant polynomial equation during transmission from the client terminal to the server, [RSA-4096](https://en.wikipedia.org/wiki/RSA_(cryptosystem)) can be used to encrypt that polynomial equation in the client, and decrypt it in the server.

For instance:

1. The polynomial equation, being a special data format, can use a highly secret and obfuscated algorithm, which is embedded in the source codes of both the client and server software, to be encoded into numbers and number separators
2. The client terminals can use the [RSA-4096](https://en.wikipedia.org/wiki/RSA_(cryptosystem)) public key, which will be updated periodically from the server, to encrypt those numbers
3. The client sends the encrypted version to the server
4. The server can use the [RSA-4096](https://en.wikipedia.org/wiki/RSA_(cryptosystem)) private key, which will have coordinated updates with all client terminals, to decrypt those numbers
5. The server can use the same algorithm as the client terminal counterpart to decode those numbers and number separators to recover the polynomial equation

Similarly, for the digital certificate acting as [salts](https://en.wikipedia.org/wiki/Salt_(cryptography)) of the roots of the resultant polynomial equation, [AES-256](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard) can be used:

1. The digital certificate, being a special data format, can use a highly secret and obfuscated algorithm, which is embedded in the source codes of the client software, to be encoded into numbers and number separators
2. The client terminals can use the [AES-256](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard) key, which is embedded in the source codes of the client software in a highly obfuscated manner, to encrypt those numbers
3. The client sends the encrypted version to the server and the server stores that encrypted version
4. When the client terminal attempts a login, the server sends the encrypted version of the digital certificate back to the client terminal
5. The client terminal can use the same [AES-256](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard) private key to decrypt those numbers
6. The client can use the same algorithm for encoding the digital certificate to decode those numbers and number separators to recover the original digital certificate

As for the [salts](https://en.wikipedia.org/wiki/Salt_(cryptography)) of the password in plaintext to be [hashed](https://en.wikipedia.org/wiki/Cryptographic_hash_function), the same flow applies:

1. The client terminals can use the [AES-256](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard) key, which is embedded in the source codes of the client software in a highly obfuscated manner, to encrypt the [salt](https://en.wikipedia.org/wiki/Salt_(cryptography))
2. The client sends the encrypted version to the server and the server stores that encrypted version
3. When the client terminal attempts a login, the server sends the encrypted version of the [salt](https://en.wikipedia.org/wiki/Salt_(cryptography)) back to the client terminal
4. The client terminal can use the same [AES-256](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard) key to decrypt the encrypted version and recover the original [salt](https://en.wikipedia.org/wiki/Salt_(cryptography))

**Summary**

The whole password setup/change process is as follows:

1. The client inputs the user ID and its password in plaintext
2. A [salt](https://en.wikipedia.org/wiki/Salt_(cryptography)) for [hashing](https://en.wikipedia.org/wiki/Cryptographic_hash_function) the password in plaintexts will be randomly generated
3. The password will be combined with a fixed [pepper](https://en.wikipedia.org/wiki/Pepper_(cryptography)) in the client software source code and the aforementioned [salt](https://en.wikipedia.org/wiki/Salt_(cryptography)), to be [hashed](https://en.wikipedia.org/wiki/Cryptographic_hash_function) in the client terminal by [SHA3-512](https://en.wikipedia.org/wiki/SHA-3) afterwards
4. The [hashed](https://en.wikipedia.org/wiki/Cryptographic_hash_function) password as a hexadecimal number with 128 digits will be converted to a base 256 number with 64 digits, which will be repeated 8 times in a special manner, and then broken down into a list of 512 literals, each being either numeric literals 1 to 100 or any of the 156 named constants
5. Each of those 512 numeric literals or named constants will be attached with existing numeric literals and named constants via different ways and combinations of additions, subtractions, multiplications and divisions, and the whole attachment process is determined by the fixed [pepper](https://en.wikipedia.org/wiki/Pepper_(cryptography)) in the client software source code
6. The same attachment process will be repeated, except that this time it’s determined by a randomly generated [salt](https://en.wikipedia.org/wiki/Salt_(cryptography)) in the client terminal
7. That list of 512 distinct roots, with the ordering among all roots and all their literal expressions preserved, will produce the resultant polynomial equation of degree 512
8. The resultant polynomial equation will be encoded into numbers and number separators in the client terminal
9. The encoded version will be encrypted by [RSA-4096](https://en.wikipedia.org/wiki/RSA_(cryptosystem)) on the client terminal with a public key there before being sent to the server, which has the private key
10. The server decrypts the encrypted polynomial equation from the client with its [RSA-4096](https://en.wikipedia.org/wiki/RSA_(cryptosystem)) private key, then decode the decrypted version in the server to recover the original polynomial equation, which will finally be stored there
11. The 2 aforementioned different salts will be encrypted by 2 different [AES-256](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard) keys in the client software source code, and their encrypted versions will be sent to the server to be stored there
12. The time complexity of the whole process, except the [SHA3-512](https://en.wikipedia.org/wiki/SHA-3), [RSA-4096](https://en.wikipedia.org/wiki/RSA_(cryptosystem)) and [AES-256](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard), should be controlled to [quadratic time](https://en.wikipedia.org/wiki/Time_complexity#Polynomial_time)

The whole login process is as follows:

1. The client inputs the user ID and its password in plaintext
2. The client terminal will send the user ID to the server, which will send its corresponding [salts](https://en.wikipedia.org/wiki/Salt_(cryptography)) for [hashing](https://en.wikipedia.org/wiki/Cryptographic_hash_function) the password in plaintexts and forming distinct roots respectively, already encrypted in [AES-256](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard) back to the client terminal, assuming that the user ID from the client does exist in the server(otherwise the login fails and nothing will be sent back from the server)
3. The password will be combined with a fixed [pepper](https://en.wikipedia.org/wiki/Pepper_(cryptography)) in the client software source code, and the aforementioned [salt](https://en.wikipedia.org/wiki/Salt_(cryptography)) that is decrypted in the client terminal using the [AES-256](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard) key in the client software source code, to be [hashed](https://en.wikipedia.org/wiki/Cryptographic_hash_function) in the client terminal by [SHA3-512](https://en.wikipedia.org/wiki/SHA-3) afterwards
4. The [hashed](https://en.wikipedia.org/wiki/Cryptographic_hash_function) password as a hexadecimal number with 128 digits will be converted to a base 256 number with 64 digits, which will be repeated 8 times in a special manner, and then broken down into a list of 512 literals, each being either numeric literals 1 to 100 or any of the 156 named constants
5. Each of those 512 numeric literals or named constants will be attached with existing numeric literals and named constants via different ways and combinations of additions, subtractions, multiplications and divisions, and the whole attachment process is determined by the fixed [pepper](https://en.wikipedia.org/wiki/Pepper_(cryptography)) in the client software source code
6. The same attachment process will be repeated, except that this time it’s determined by the corresponding [salt](https://en.wikipedia.org/wiki/Salt_(cryptography)) sent from the server that is decrypted in the client terminal using a different [AES-256](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard) key in the client software source code
7. That list of 512 distinct roots, with the ordering among all roots and all their literal expressions preserved, will produce the resultant polynomial equation of degree 512
8. The resultant polynomial equation will be encoded into numbers and number separators in the client terminal
9. The encoded version will be encrypted by [RSA-4096](https://en.wikipedia.org/wiki/RSA_(cryptosystem)) on the client terminal with a public key there before being sent to the server, which has the private key
10. The server decrypts the encrypted polynomial equation from the client with its [RSA-4096](https://en.wikipedia.org/wiki/RSA_(cryptosystem)) private key, then decode the decrypted version in the server to recover the original polynomial equation
11. Whether the login will succeed depends on if the literal expression of the polynomial equation from the client exactly matches the expected counterpart already stored in the server
12. The time complexity of the whole process, except the [SHA3-512](https://en.wikipedia.org/wiki/SHA-3), [RSA-4096](https://en.wikipedia.org/wiki/RSA_(cryptosystem)) and [AES-256](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard), should be controlled to [quadratic time](https://en.wikipedia.org/wiki/Time_complexity#Polynomial_time)

For an attacker trying to get the raw password in plaintext:

1. If the attacker can only sniff the transmission from the client to the server to get the encoded then encrypted version(which is then encrypted by [RSA-4096](https://en.wikipedia.org/wiki/RSA_(cryptosystem))) of the polynomial equation, the [salt](https://en.wikipedia.org/wiki/Salt_(cryptography)) of its roots, and the counterpart for the password in plaintext, the attacker first have to break [RSA-4096](https://en.wikipedia.org/wiki/RSA_(cryptosystem)), then the attacker has to figure out the highly secret and obfuscated algorithm to decode those numbers and number separators into the resultant polynomial equation and the way its roots are attached by existing numeric literals and named constants
2. If the attacker has the resultant polynomial equation of degree 512, its roots must be found, but there’s no direct formula to do so analytically due to [Abel-Ruffini theorem](https://en.wikipedia.org/wiki/Abel%E2%80%93Ruffini_theorem), and factoring such a polynomial with 156 different named constants efficiently is very, very complicated and convoluted
3. If the attacker has direct access to the server, the expected polynomial equation can be retrieved, but the attacker still has to solve that polynomial equation of degree 512 to find all its roots with the right ordering among them and all their correct literal expressions
4. If the attacker has direct access to the client software source codes, the [pepper](https://en.wikipedia.org/wiki/Pepper_(cryptography)) for [hashing](https://en.wikipedia.org/wiki/Cryptographic_hash_function) the password in plaintext, the [pepper](https://en.wikipedia.org/wiki/Pepper_(cryptography)) used on the polynomial equation roots, and the highly secret and obfuscated algorithm for using them with the [salt](https://en.wikipedia.org/wiki/Salt_(cryptography)) counterparts can be retrieved, but it’s still far from being able to find all the roots of the expected polynomial equation of degree 512
5. If the attacker has all those roots, the right ordering among them and all their correct literal expressions still have to be figured out, and the [salts](https://en.wikipedia.org/wiki/Salt_(cryptography)) and [peppers](https://en.wikipedia.org/wiki/Pepper_(cryptography)) for those roots has to be properly removed as well
6. If the attacker has all those roots with the right ordering among them, all their correct literal expressions, and [salts](https://en.wikipedia.org/wiki/Salt_(cryptography)) and [peppers](https://en.wikipedia.org/wiki/Pepper_(cryptography)) on them removed, the attacker has effectively recovered the [hashed](https://en.wikipedia.org/wiki/Cryptographic_hash_function) password, which is mixed with [salts](https://en.wikipedia.org/wiki/Salt_(cryptography)) and [peppers](https://en.wikipedia.org/wiki/Pepper_(cryptography)) in plaintext
7. The attacker then has to figure out the password in plaintext even with the [hashing](https://en.wikipedia.org/wiki/Cryptographic_hash_function) function, [salt](https://en.wikipedia.org/wiki/Salt_(cryptography)), [pepper](https://en.wikipedia.org/wiki/Pepper_(cryptography)), and the highly secret and obfuscated algorithm that combines them known
8. Unless there are really efficient algorithms for every step involved, the time complexity of the whole process can be as high as [factorial time](https://en.wikipedia.org/wiki/Time_complexity#Factorial_time)
9. As users are still inputting passwords in plaintexts, [dictionary attacks](https://en.wikipedia.org/wiki/Dictionary_attack) still work to some extent, but if the users are careless with their [password strengths](https://en.wikipedia.org/wiki/Password_strength), then no amount of cryptography will be safe enough
10. Using [numerical methods](https://en.wikipedia.org/wiki/Root-finding_algorithms) to find all the roots won’t work in most cases, because such methods are unlikely to find those roots analytically, let alone with the right ordering among them and all their right literal expressions, which are needed to produce the resultant polynomial equation with literal expressions exactly matching the expected one
11. Using [rainbow tables](https://en.wikipedia.org/wiki/Rainbow_table) won’t work well either, because such table would be way too large to be used in practice, due to the number of polynomial equations with degree 512 being unlimited in theory
12. Strictly speaking, the whole password encryption scheme isn’t a [one-way function](https://en.wikipedia.org/wiki/One-way_function), but the time complexity needed for encryption compared to that for decryption is so trivial that this scheme can act like such a function

Areas demanding further researches:

1. The time complexity for factoring a polynomial of degree n with named constants into n factors analytically
2. Possibilities of [collisions](https://en.wikipedia.org/wiki/Collision_(computer_science)) from the ordering among all roots and all their different literal expressions
3. Existence of efficient algorithms on finding the right ordering among all roots and all their right literal expressions
4. Strategies on setting up the fixed [peppers](https://en.wikipedia.org/wiki/Pepper_(cryptography)) and generating random [salts](https://en.wikipedia.org/wiki/Salt_(cryptography)) to form roots with maximum encryption strength

Essentially, the whole approach on using polynomial equations for encryptions is to exploit equations that are easily formed by their analytical solution sets but very hard to solve analytically, especially when exact literal matches, rather than just mathematical identity, are needed to match the expected equations.

So it’s not strictly restricted to polynomial equations with a very high degree, but maybe very high order partial differential equations with many variables, complex coefficients and functions accepting complex numbers can also work, because there are no known analytical algorithm on solving such equations yet, but analytical solutions are demanded to reproduce the same partial differential equations with exact literal matches, as long as performing partial differentiations analytically can be efficient enough.