Suggested Into Paragraph 1 - 3

Using quantum mechanics inline with binary or trinary data can allow for dynamic compression and data mixing without knowing any information other than what's in the string along with standardized procedures. Procedures involved can be used in a combination of multiple processes, all of which can work stand-alone, but in conjunction become an environment to perform quantum manipulations. Quantum-manipulated data can be smart data or self-directing data, which is data that contains the way it is to be used by being able to wrap extra data with the base data, like APIs.

Because everything is deterministic, we'll use mathematics & old-fashioned work to add entropy, which will inflate our output but will also help reduce the ability to crack or break a Super Positioned Text (SuPosXt [sue – pose – it] or SPX) hash while giving the keys to check accuracy during detangling. We will first look at data mining so we can build patterns for token swapping later. Then we will prepare the data for graphing based on complexities. Shift the data to a non-future state before performing checks to prepare for our first set of token swaps from the built patterns to remove added data & graphing boundaries. Finally we'll perform quantum entanglements to finally compress the graph into a type of hash.

If all goes to plan, we'll be able to perform various manipulations to the data that should be reproducible, similar to how quantum states are reproducible, making the resulting quantum-manipulated data technically reversible.

Data-Mining Token Patterns

Using tokens to represent sections of data in a controlled manner can allow us to take the 13 smallest patterns in binary to create a predetermined sequence of binary patterns. We can use this same technique to obtain predetermined sequences from any form of data structure, for example, binary, trinary, CRC-encodings, & secure-hashes. If our 13 smallest patterns are also the most commonly seen patterns, like for non-binary or non-trinary strings, then we can assimilate this base key for any data type.

We need to build or have ready 3 alphabet reference arrays; they do not have to be alphabet characters, but they do need to be a set limit of the max allowed length to swap for a token. For

example, the base SPX system uses the English alphabet, so the upper limit is 26 characters to be swapped per token. The first alphabet array will need to be in one case and contain the full limit so it can be a “Full Reference Array.” The second alphabet array will need to be 13 less than the max but in the opposite case of the first alphabet array, so this array will be the “2nd Reference Array.” Then finally, the third alphabet array is only 5 characters long, also in the opposite case of the first alphabet array, to be our “3rd Reference Array.” Normally we will add “0” & “1” to the far left of all arrays. However, we do not add the “1” in the “ones” position for alphabet reference arrays (except for the 3rd reference array); they will only get “0” added to the furthest left side of the array. Any and all exceptions to the 1/0 rule (placing “0” & “1” values in the “zero” & “ones” places of the arrays) will be mentioned as they appear. We do not want to use the first character in the alphabet reference array, for that is reserved for single-character token swaps. This is why the third alphabet reference array has the “1” in the “ones” position (full 1/0); it has no “a” or first alphabet character in its array. Below are the example reference arrays:

1. ["0", "A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", “O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"]
2. ["0", "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n"]
3. ["0", "1", "o", "p", "q", "r", “s"]

The first two arrays have only a half 1/0, while the third has a full 1/0, since there is no “1st character of the alphabet referenced” in that array. The remaining alphabet characters missing in the opposite case of the first alphabet array are used as token type identifiers. From our example, the token type identifiers are “t,” “u”, “v”, “w”, “x”, “y”, “z”. We’ll assign these later. The alphabet reference arrays and type identifiers that share the same reference don’t have to be an alphabet, but these arrays and identifiers can be anything, including any pattern(s) that may be seen in the array. The important part is the length of the first array is the full length of the last 2 arrays & their token type identifiers, while everything in all the arrays and identifiers should be equivalent but different from state or case. So the way we can explain this mathematically is by formula, where “Arr#” is the numbered array and [IDs] is the group of identifiers. These formulas are seen after the following infographics:

Array Reflection Wheel

The "Array Reflection Wheel" is a visual representation of how all the arrays reflect from the 1st reference array.

Arr1 = Arr2 + Arr3 + [IDs]

Example Array Formula (non-specific)

Arr0 = [a-z]26

Example Arr0

[A->Z]26 = [a->n]14 + [o->s]5 + [t->z]7

Example array formula (previous example specific)

A good example of a non-alphabetic full reference array is using primes but raising a single prime to the next prime value. If you need to get an opposite case of this, you simply raise all of these by the prime before the initial prime in the first slot. Next is an example of primes used as a full reference array, which is as follows:

["7","1977326743","96889010407",[…],
"16357825134699997867021203570792023496606186572836247166519684004
81920745472"]

The previous example was used for dramatic effect but the point still stands and the previous example full reference array could be written in the following notation:

Arr1 = [P7->P89]20

Arr0 = [P7^5->P89^5]20

Example Arr1 & Arr0 formulas

[P7->P89]20 = [Arr1^7]7 + [Arr^3]5 + [Arr1^2]8

Example Non-Alphabetic Reference Arrays Formula

This notation uses "P" for prime, points to which prime it begins at, which it ends at, and how many primes are in the array. The opposite case of the 1st reference array is "Arr0" in notation form, which is the entire first array raised by the previous prime seen in the first slot. So Arr0 is Arr1 raised by 5. We can use this simple way to describe any changed or modified full reference array by notations. Any notation like this can be placed within the programming slots for the decoder to know of any special changes if the decoder doesn't allow for direct influence uploading of one's patterns and array formats.

Now that we have a limit set with the 1st reference array, we can build the patterns. Looking at the smallest patterns, we need to remove any single-length patterns, for they would inflate anything we apply them to. Starting at the length of 2 or L2, we will see "00", "11", "01", "10", "100", "011", "101", "010", "1001", "0110", "10100", "01011", "001101". We purposely do not use any repeating single-character patterns over L2 (or L3+) for we will use a specific pattern replacement procedure for them. The procedure we are going to use for making the larger sequences will be a similar method for single-character replacement tokens, just with more steps, which will be covered later.

If you are using an alphabet reference or any full reference over 26 (not counting the null position), then there is room to add to any set of arrays. Just ensure any new array added gets an identifier added as well. We are using the relation of identifiers to full reference for determining what the starting bits may be. For example: If you needed to change the starting bits, you would place that information within the programming slots. You would have to use notations or Wave-Data APIs to inform the decoder of the patterns if the decoder doesn't allow for direct uploading of your patterns and forms.

When building patterns, we never want to accept a new pattern that is a duplicate of a previous pattern in any array. We will generate patterns and then check if we have those patterns already; if not, we'll use them, and if so, we won't. We will use our 13 patterns from smallest to largest so we have the best chances for the smallest

considerations of patterns possible because we are only going to build up to 8 patterns per array. We will never use the null position of any of these arrays; most of these will not use the ones position either, and we will not go past 9 in position. This prevents double-digit overflow in the later processes as well as prevents duplicate identifiers. Each pattern array has a rule for its generated patterns, which are the following:

Pattern Palindromes ***[1st Generated-Pattern Array]***

For Example: ***"010010" is a pattern palindrome, specifically a palindrome with the reference pattern "010" included. "010" is the 7h pattern in our pattern array.***

Example array: ["0", "1", "001101101100", "1010000101", "0101111010", "10011001", "01100110", "101101", "011110", "010010"]

- Pattern Reversed ***[2cd Generated-Pattern Array]***

  For Example: ***"00101" is a pattern reversed, specifically a reversal of "10100" the 10th pattern in our pattern array.***

  Example array: *["0", "1", "110001101001101110", "0010101101011001", "101100101100", "1101011010", "0010100101", "101100", "11010", "00101"]*

- Pattern Palindromes PLUS it's Reversed ***[3rd Generated-Pattern Array]***

  For Example: ***"011110011110" is a pattern palindrome plus it's reversed self, specifically a reversal of the palindrome "011110", which is "011110", added together from our 1st Generated-Pattern Palindrome array.***

  Example array: *["0", "1", "100110011001100110011001", "011001100110011001100110",*

“0011011011000011011011100”, “10100001011010000101”, “01011110100101111010”, “10110110110110110101”, “01001001001001001010”, “011110011110”]

- Pattern Reversed-Doubled PLUS it’s Palindrome-Reversed ***[4th Generated-Pattern Array]***

  For Example: ***“010010010010” is a pattern reversed-doubled plus it’s palindrome self, specifically “010” was used to make first “010010” then it’s palindrome “010010” to be added in generated order for “010010010010”.***

  Example array: *[“0”, “1”, “10110010110000110100 1101”, “11010110100101101011”, “00101001011010010100”, “1001100110011001”, “0110011001100110”, “110110011011”, “101101101101”, “010010010010”]*

These example arrays are fully formatted

We have taken a single array of 13 patterns and generated 32 unique patterns. Now we order the generated patterns in each array from the largest (furthest left side) to the smallest (furthest right side), again ordered from within the array, before we add “0” & “1” to the furthest left side of each array (using the null one's position of the array). The example arrays were already set for this formation. We want to always replace the largest possible pattern first, then work our way down to the smallest patterns last while also not using “0” & “1” both in position & in value, aka ignoring the 1/0.

So far, we have forced generated 32 unique strings that should always end up the same for as long as the initial 13 key patterns are in the same order each time. For as long as the generated patterns are always largest to smallest, we will always get the largest possible compression per data-set in our soon-to-be hash. Next, we will need to generate mixing tokens to hide the differences between data sets.

The next set of arrays is going to be specific to your data-formation but the examples given will still apply to the same examples as before. We have an index which could also be an API string.

However, you need to set up 3 arrays each with at least 5 tokens inside. All three of these should always have the same number of tokens inside each plus 1/0. These tokens need to be your index keys or API key-bits, even if you have to split them up some. For example, here's our 3 index key arrays, which are as follows:

1. *[“0”, “1”, “0.001”, “-500.1”, “-505.11”, “-330.1”, “220.1”]*

1. *["0", "1", "770", "50.1", "55.01", "60.01", "0.3"]*

1. *["0", "1", "0.2", "0.4", "0.5", "1100", "111"]*

The index key arrays start off with the most complex (containing the most number of unique non-alphanumeric characters) at longer lengths, then work down to the least complex at the smaller lengths, & finally ending with no-complex tokens (alphanumeric only). The index key arrays of non-1/0 tokens can only be as long as the number of non-1/0 tokens present in the 3rd reference array.

We need 2 more arrays, one to specifically handle our data-set breaks & another to handle unique edge cases. Again, which tokens/strings you need in your case may change, but here are the base examples in line with the previous examples so far:

1. *["===;", "==;1;0;", "==;0;1;", "==;2;", "=;1;0;", "=;0;1;", "=;2;", ";2;", ";1;", ";0;"]*

1. *["==", "=", ";1", ";0", "1;", "0;", ";.", ";", ".", “-"]*

The data-break array attempts to remove all data breaks from most generic-complex (most number of non-alphanumeric characters at the longest lengths) to the least generic-complex (least number of non-alphanumeric characters at the shorter lengths), always putting non-alphanumeric tokens before alphanumeric-containing tokens. The unique edge case array is also set up in the same manner as the data-break array. These arrays are similar to the reference arrays where we may use the “1” in position & in value, but these ones also may use “0” in reference but not in value to ensure we can use the full 10 slots of the array. This will give us a unique token, “null-soft.” It's a null-pointer token, meaning we will see a “0” before the token identifier. These are the only tokens possible to have a null-soft token.

Now that we know what we are wanting to replace or use as token bases, we may need to search for first duplicates & sequences of these patterns in specific ways and then work our way down to just these arrays to always get the largest possible compression.

Binary VS Faux-Binary States

The majority of our data will need to be either in a binary or faux-binary state. Trinary does count as a binary state, but your system will need to be handled more specifically if trinary is used; otherwise, this will go over using a binary state or a faux-binary state for your data.

The advantage of simply using the binary state of your data is that it is going to have the smallest possible compression, and oftentimes you won't need an index at all (if all binary lengths are equal). The index dataset can be, like all parts of the data-sets, removed or swapped out for a similar but different data-sets. The tokens used for compressing that data-set may need to be adjusted, or the data in the data-set will need to be modified to work as an index data-set. The advantage of using a faux-binary state is simply privacy. A faux-binary state may reduce overall binary characters seen, but the true point of faux-binary is privacy in what you send, because faux-binary states don't have to contain the actual data but enough data to rebuild to the original state.

A faux-binary state is taking your data and modifying it to appear like a binary state. The End-to-End Encryption font, Node, is designed to take ASCII characters and turn them into a physically encrypted faux-binary output. Based on the complexity states of simple node-connection configurations, Node takes a character and generates a formula with binary pointers to then create a heat map to direct how to align this representative shape of the node-connection configuration to a 17-box grid. Each heat map comes with an index value to allow for decoding through reconstruction of the graph at a later time. Next is the example Node configuration table, which is read as: Character = Node_heat_map @node_index.

NODE CONFIGURATION TABLE

In each box you will see a character that may be seen as equal to its heat map with adhered data at its index code. Within the heat map you will see up to 4 characters “A” through “D”; these are the rows of the grid, while the numbers are the nodes on/off positions. The exception to this is D; D actually refers to the aligning box (the 17th box in the grid) or under-table row. This is done specifically to confuse others. The entire system is based on mapping these “nodes” to a grid, but in that mapping you never need to know the last row, as long as you have the rest of the information. So we take advantage of this situation and purposely ignore and never use the real 4th row in the grid in reference. When we rebuild the pattern based on that grid, it will still work for anything going to the fourth row of the grid; it must either pass through the center or start/end on a different row. Even if a 3 or 2 straight was on the 4th row on its own, we would see its index code and be able to work backwards what is missing using the checks to find the blank space that is missing data (we already do this for blank spaces in general). We may also have data residue that should be the missing bits or some of the missing bits from the mapping.

If you only use binary data, then you will need to divide the binary string into groups of 4 or groups with lengths of 4 (aka L4). In each L4 group: the 1st position is data-rail A, 2cd position is data-rail B, 3rd position is data-rail C, 4th position is data-rail D. The index is simply the length of that binary data set. For example, if your binary set is a full length of 8 (L8), its index would be “0.8”. However, if your binary set is a full length of 16 (L16), its index would be “1.6”, so you could only have a max limit of L99, which would be an index of “9.9”. Just like the Node Configuration table, each input should be handled separately, but to save space in the index field, this extended binary consideration of indexing could be used.

Dimensional Shifting

Once our data is set up in the graph where the raw data bits are in 4 or fewer rows, we can begin shifting them. After we shift the bits, we’ll run everything through a specially modified formula to obtain a parity check output or action bit. This action bit can be compounded and used to help ensure no data was changed between sends.

To shift the data we have, we first look at the first bit in a row and record that bit. Then we compare this [previous] bit to the next bit; if they are the same, we'll record a "1," but if they are different, we will record a "0." If you are using a trinary system, you will use a "2" if the previous bit is more than the next bit, while a "0" would represent the previous bit being less than the next bit. We will then compare the new previous bit to the new next bit and continue until the entire row has been compared and recorded. Do this for all binary-like state rows you have.

Shifting binary-like data allows you to only have 1 bit per row of real data being transferred. We're never actually sending the data but an "overflowed vector" of the data, while in this case we are not overflowing a buffer but buffering the difference between two bits. To reverse this shift, we start at the first bit, record it, then compare the now [previous] bit to the next bit and work accordingly in the manner as before. There are no identifiers for what form of state is used, so a quick check down the string of each row should be down for "2" and if none is found, a binary or binary-like state should be assumed.

We will take the end result of the data shifting and toss each bit through a Modified Futurama Theorem (MFT) formula, adding the previous output of the MFT (if there was one) beforehand and recording the last bit of output from the MFT. This last output bit should be our action bit. We can perform a traditional hamming code parity check at this point if we want to verify ourselves. As of writing this, the current SPX demo does use traditional hamming code parity checks to verify itself. All parity checks are recorded in some form in their designated data-sets (determiner slots).

The Modified Futurama Theorem is as follows:

Ceiling(Trunc(input_SUM * 3.14) + Absolute(((current_position * 18) + input_length) - 1) + MOD) % 2

CAP(TRUNC(INPUT _SUM* PI) + ABS(((POSITION * MAX_POSITION) + INPUT_LENGTH) - 1) + ADJUSTER) % MOD

The MFT uses the input sum (ABSOLUTE((full_action_counter * max_loop_length) + input_length) - 1) multiplied by PI (3.14) to get a radius based on that value to then keep it at its own length and keep the output at mod2 or nothing higher than the digit "1." This will always shift the data into a form equal to its length, and in this case we are only dealing with a length of 2, and our end-modified allowed output is also capped at 2. This keeps all of our data in that binary formation. If we used trinary, we would need to change the end modifier to a "3" instead of a "2".

The MFT can also be adjusted, so for example, when we look to see if the total of 1's seen in the previous parity bits as a whole is even and if they are, then we use a "+2" adjuster instead of "+MOD", otherwise we use a "+1" adjuster instead of "+MOD" and this will force the output to always be properly formed and without time-based adjustments. To make time-based adjustments, change the "+MOD" to a value that is exponentially higher than the "+MOD"

adjuster. So, if we want to go 1-step in the “future”, we will use “MOD^2”. The 2-steps would be “MOD^3”. Where “MOD^1” is the assumed natural starter position for the “+MOD” adjuster. NOTE: Time-Adjusting to the future/past is mathematical guessing based on the formula presented, so the real-future or past parity output may be different, but this could be used to guess a possible parity without the extra data involved.

Spherical-Parallelogram Check

Why just trust multiple parity checks when you can verify a parity check. One of the key bits of information we want to superposition in one of our data-sets is a “Magic Number”, which is a derivative of amalgamated data based on wrapping a parallelogram around a sphere. We will build a parallelogram and sphere based on the length of input, turning this into a form of check based on what we should obtain after the detangling process. But we also base parts of the parallelogram’s & sphere’s data points off the number of internal loops that need to be made for the completion of rebuilding the graph as well as the input sum. For example, formulas, we will only be using JavaScript Math notation.

Define the starting variables:

- **z =** *Input_Sum = (ABSOLUTE((full_action_counter * max_loop) + Input_length) - 1)*
- **y =** *Current_Micro-Loop_Position*
- **x =** *Input_Length (or Number_of_Seen_Inputs)*
- **[ANGLE] a =** *(((y * 18) + x) -1)*
  - ***18 is our loop limit***

Find the parts of the parallelogram:

- **A|C =** *Round((trunc[z * PI]) + ((a) % 2)^((PI + z) / (a)))*
- **B|D =** *Round((180-(trunc[z & PI]) + ((a) % 2)^((PI + z) / (a)))*
- **[ANGLE] b =** *Round(a / sin([round-up{trunc[z * PI]} + ((a) % 2)^((PI + z) / (a))]))*
- **[HEIGHT] h =** *Round(a * (a / sin([round-up]{trunc[z * PI] + ((a % 2)^((PI + z) / a))*sin([round-up]{(trunc[z * PI] + a % 2)^((PI + z) / a)})}))*

If output is negative, drop the negative sign

- **P =** *Round((a * 2) + (a / sin([round-up]{trunc[z * PI] + ((a % 2)^((PI + z) / (a * 2)))})))*
- **AREA =** *Round((1/2)*(a /(sin([round-up]{trunc[z * PI] + ((a % 2)^((PI + z) / ((a * (a * (a /sin([round-up]{trunc[z * PI] + ((a % 2)^((PI + z) / ((a * (sin([round-up]{a}])))))})))))))})))))*

Find the parts of the sphere:

- **[SPHERE] VOLUME =** *Round((4/3) * Pi * POW(trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / (a * (sin([round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / a))})))){drop any negative sign at this point} / 2, 3))*
- **[SPHERE] RADIUS =** *Round(a * (a / (sin([round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / (a * (sin([round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / (a)))})))))})))){drop any negative sign at this point} / 2*

Form the "*Magic Number*":

- **MAGIC NUMBER =** *Round(Round(Round(((4/3) * Pi) * POW(Round(a * (a / sin([round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / (a * (sin([round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / (a)))})))))})))){drop any negative sign at this point} / 2, 3)) % ((1/2) * (a / (sin[round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / (a * (a * a)) / (sin([round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / a))})))))})))) /*

*Round(a * (a / (sin([round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi +z) / (a * (sin([round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / a))}))))))}))) {drop any negative sign at this point} / 2))*

At this point, we should have the parallelogram and sphere with values that should be capable of being mapped in a 3D space that would allow the parallelogram to wrap around the sphere to touch itself but not overlap itself. Taking the Magic Number or by finding the difference between the parallelogram's volume and the sphere's volume (dropping any negative sign you may receive), you can have up to 2 different numbers that can be rediscovered once you are able to map the objects to the 17-grid graph.

Pattern-Based Token Swaps

We already built the patterns for swapping; now to go over how the entanglement tokens work. These will be the base steps to create all the Pattern-Based tokens for swapping. We will also go over the specific methods to look for sequences of patterns to swap for as well.

The tokens we are going to use will be NUMBER (+) IDENTIFIER (+) REPEAT_ID. This is the generic setup for all pattern-based tokens because it will entangle a small amount of information into the token. For example, the identifier tells us which unique pattern array we are using, and what's in that array can be anything. The same goes for number & repeat ID; anything can be used to represent these values. Starting off by simply looking for the predetermined pre-generated tokens (the first 4 generated arrays): The number we are looking for is the position of the token in the array; the identifier is one of the remaining lower-case characters [t-z]; the repeat ID is the number of times we saw this pattern repeated in a row as equal to its position in the full reference array.

The example identifiers are as follows:

"**w**" is the identifier for the *1st Generated-Pattern Array*

"**x**" is the identifier for the *2cd Generated-Pattern Array*

"**y**" is the identifier for the *3rd Generated-Pattern Array*

"**z**" is the identifier for the *4th Generated-Pattern Array*

"**t**" is the identifier for the 1st Index Keys Array

"**u**" is the identifier for the 2cd Index Keys Array

"**v**" is the identifier for the 3rd Index Keys Array

"***a***" is the identifier for the *Data-Break Array*

"***A***" is the identifier for the *Unique Edge Cases Array*

The way we chose these identifiers is the last 3 reference tokens in the full reference array, but in the opposite case, it is always for the first 3 generated-pattern arrays, while the first 3 available tokens after the 3rd reference array are always used for the index keys arrays, and the first token of reference for both cases is always double used, once for the data break and unique edge cases but also for single token replacements (although single token replacements only use the case used in the full reference array).

The generated-pattern tokens are pretty straightforward; the number is the array position for the token to put back, the letter immediately after the number is the identifier, and if there is an opposite case letter after the identifier, then that is the number of times to loop the token for a full replacement. Determining the repeating single-character tokens is again fairly straightforward. We will only look for repeating "0" and "1" (or also repeating "2" for trinary-based systems). So we will place a single bit of the repeating single-character bits that is being swapped for plus the capital-case character from the full reference array, which represents the number of times the single bit was seen in a row. So if "000000" was replaced with "0F," we'd replace the "F" for "00000" to make "000000" during reversal. We will only replace L3+ repeating single-character strings (or only strings with lengths of 3 or more), because "00" and "0B" are the same length, while "0" turned to "0A" only inflates, so that's pointless extra work.

The Index Keys tokens are less straightforward. Whereas the generated-pattern tokens are simply what's already been swapped, these index key tokens are first mixed with combinations of one another & then a simple token swap as before.

The goal is again to find the largest possible swaps first. At some point we need a way to determine edge cases, similar to before. So, we will introduce single tokens to be used as modifiers for these last token sets. This "7th array" is composed of the most commonly seen single-character tokens in the index data set that isn't the 1/0 set. Your 7th array may differ, but you will want to make sure no alphanumerical character is aligned by value to its position, so if your 7th array token is "3", make sure it's not in the 3rd position of the array.

Now we will list the complexities used to create the patterns in order to do them in during this phase of token swapping. And they are as follows (MIXING_ARRAY_REFERENCE (+) IDENTIFIER(S) (+) REPEAT_ID):

LOWtuvCAP: *the initial lower-case character position equals the position for the token from the "t", "u", "v" arrays to swap back in the order shown (**t-u-v**) and finally the*

*capital-case character position is equal to the number of times seen (**if seen only once, no capital is needed**).*

LOW7tCAP: *the initial lower-case character position equals the position for the token from the next two arrays; the "7th array", **["0", "1", "7", "6", "5", "3", "2"]**, plus the 1st Index Keys Array and finally the capital-case character position is equal to the number of times seen (**if seen only once, no capital is needed**).*

LOW7uCAP: *the initial lower-case character position equals the position for the token from the next two arrays; the "7th array", ["0", "1", "7", "6", "5", "3", "2"], plus the 2cd Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).*

LOW7vCAP: *the initial lower-case character position equals the position for the token from the next two arrays; the "7th array", ["0", "1", "7", "6", "5", "3", "2"], plus the 3rd Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).*

LOWLOWtCAP: *the initial 2 lower-case characters positions equals the positions for the next two tokens, with the 1st lower-case character for the 1st token & the 2cd for the 2cd token for replacement from the 1st Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).*

LOWLOWuCAP: *the initial 2 lower-case characters positions equals the positions for the next two tokens, with the 1st lower-case character for the 1st token & the 2cd for the 2cd token for replacement from the 2nd Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).*

LOWLOWvCAP: *the initial 2 lower-case characters positions equals the positions for the next two tokens, with the 1st lower-case character for the 1st token & the 2cd for the 2cd token for replacement from the 3rd Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).*

LOWtCAP: *the initial lower-case character position equals the position for the token from the next array, the 1st Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).*

LOWuCAP: *the initial lower-case character position equals the position for the token from the next array, the 2nd Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).*

LOWvCAP: *the initial lower-case character position equals the position for the token from the next array, the 3rd Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).*

The data-break tokens are very straight forward, NUM (+) "a". The number is the position of the token on the data-break array, while the "a" is the identifier, so we know where to look for the token. We won't need to look for anything else with data-breaks, but in the last array we'll need to mix things up a bit more.

The unique edge case tokens are going to use 3, then 2, then 1 sets of tokens in a row. Similar to how we had the LOWtuvCAP, we'll combine various tokens in the same array, the edge cases array, instead of different arrays (t-u-v). Next is a list of the following complexities to find the edge cases, which is as follows:

LOWLOWLOWCAP: *Each lower-case character represents the location of the token to be placed here from the array, the Unique Edge Cases Array, while the capital-case character position is equal to the number of times seen (if seen only once, capital is needed).*

LOWLOWCAP: *Each lower-case character represents the location of the token to be placed here from the array, the Unique Edge Cases Array, while the capital-case character position is equal to the number of times seen (if seen only once, capital is needed).*

LOWACAP: The *lower-case character represents the location of the token to be placed here from the next array, the Unique Edge Cases Array; while the capital-case character position is equal to the number of times seen (if seen only once, DO NOT REPLACE).*

LOW7CAP: The *lower-case character represents the location of the token to be placed here from the "7th Array", ["0", "1", "7", "6", "5", "3", "2"]; while the capital-case character position is equal to the number of times seen (if seen only once, DO NOT REPLACE).*

This will increase the count by 1 if only two 7th array tokens are in a row, **L3+** */or/ "***No Low7C***" for best results*

NUMA: The *number represents the location of the token to be placed here from the Unique Edge Cases Array (NO REPEATS CONSIDERED).*

Quantum Entangled Compression

Now that we have much of the unique data entangled, swapped and removed, we should have determinable data with possible duplicate strings made within the full string. Taking advantage of any data set, hash or string that has duplicate-character output, possible could have quantum entangled tokens to help compress the string. Quantum Entangled Compression (QEC) will be very similar in method to the tokens being generated for swapping from earlier but will 100% relay on what's in the string, the length of the string & the length of our full reference array.

Let's start at finding the tokens we can reference for replacement, aka replacement tokens. Finding the replacement tokens is a matter of sectioning the string into smaller parts, even if smaller parts overlap one another, just as long as one character is different from the overlapping parts around it. This is a form of "Proof-of-Work" commonly seen in machine-verifying databases & machine-verifying web-technologies.

We will start by looking at the head or front of the string (furthest left side). We will begin by recording what we see to the limit of the shortest of either the full reference array (minus the "0") or half of the total string length. This may seem like a strange limitation, but if we consider that we are going to reference one part of the string to replace another part, then we only need to look at half the total string length, but if that half of the string length is greater than the limit of our full reference array, then it is all the length we need. So now we need to record all that we see moving over only 1 position each time, until we see a blank space in our recording string. When we get the point of having blank space, we can stop & begin limiting how much we see by 1. So, for example, if we were looking at the full reference length of 26 and when we get to the end, so it begins to have empty space, we just return to the start of the full string and look at the 25 length strings, then when seeing blanks reappear we return again to the beginning of the full string and reduce by one again, over and over. So we reduce by one after every pass through the string until we are looking for L4 bits. This will ensure we have all possible tokens viewable, and

they go from biggest to smallest (in length reference), without having to generate any tokens we won't be able to use. This is why we don't look at L3 nor L2 bits, because that would be extra work as referenced in the previous token swapping section. So we can have a single array with all these "tokens" to swap out. Remember not to accept duplicate tokens just as before.

Now, we have to look for these tokens. To do this, we are going to search the string for our tokens (one at a time) and if the token search finds results over 1, it replaces the seen tokens after the first viewed one. This ensures the original token is still there for reference and the duplicates are swapped with a QEC token. We just have to do this for all the tokens placed in the array.

A Quantum Entangled Compression (QEC) token is a specially formed token that uses our full reference array and some minor logic. The only difference between a pattern-based token and a QEC token is that the QEC token relies on logical considerations instead of referencing an array. So the QEC tokens only need the string it's compressing. The QEC tokens are 2-3 slots long, with each slot handling a different pointer; this is identical to the pattern-based tokens.

The first slot will be the full reference token that represents the location on the string. Because we can have a string larger than our upper limit, we naturally divide the string into a soft upper limit. In our case, the soft upper limit is the shortest of either the length of the full reference array or half the length of the string presented. So, as we perform the first slot considerations, we will loop through the groups but still assign the token-object found in the full reference array. For example, if we have 2 groups and both groups have a token-swap object seen at the 14th position, both will have an "N" in their 1st QEC token slot. There may be more than one formatted object in the 1st slot, so if you use characters in slot 1, you can use numbers in slot 1 before the character to show the group number. If you have over 9 groups, use double singles to showcase the groups as sections per 9. For example, if your total number of groups was 18, you could have 2 sets of 9 groups, i.e., 99[…] or 94[…]. This works because the highest digit is always before the lowest digit, so we can know they are acting as a single bit in correlation to the character bit behind it. NOTE: Using group number string-bits can inflate the end result.

The second slot will be the full reference array but in the opposite case. So if you used upper-case in slot 1, you'll use lower-case in slot 2. This means we only need one full reference array to build the other reference arrays but also this opposite case array. The object in the 2cd slot simply refers to the length of the token-object that was swapped out. So the first slot says, "Here's where the token begins," while the second slot says, "Here's how long the token is."

The optional third slot will be the full reference array again, but this time it is used as a "Repeat ID." If we can repeat the token-object in a row and find that in the string after the first time we saw the token-object, we can replace all the extra portions of that loop by referencing the original smallest pattern and then saying, "How many times to repeat that pattern," for the full return. This is also the only way we can replace part of a token object. So if we find a token-object within one of our token-objects, this replacement method would lower the length of that full reference that other tokens are pointing to. And this form of replacement has to be done last to keep confusion out of the way. Which means when detangling, you'll need to consider these first or at least check for these collisions first.

If we always do the 2-slot QEC tokens before we do the 3-slot QEC tokens, we should have reversible results, just like if we have 9 before the connecting bits, we can have multi-part strings working as a single bit. However, If we do the 3-slot QEC tokens before we do the 2-slot QEC tokens or if we order our multi-part strings with the lowest value before the 9s, we may not be able to reverse the end result, similar to a secure hash but more like a "semi-secure-hash." QEC tokens are unique if they stick to the rules set for them even though we are picking and choosing the natural entanglements we can see. Using this 3-slot token replacement method, we ensure that no matter the pattern, the process is the same, and reversal is up to the decoder & encoder.

Token Map

QEC Tokens always rely on CHARACTERS or the full reference array in two cases/states while the Non-QEC (or pattern-based) Tokens may begin alphanumerically then jump to full reference array in it's natural state, this is majorly how we tell the difference between both token sets

Before showcasing the proofs through example, let's consider the rules to use quantum entanglement for compressing data.

Starting with the token used to replace sections within a string or data-set should use the same or similar but reflective objects when compared to the objects in the other slots, rather than token-objects or alphanumeric-objects. The token used for swapping should be a 2-slot or 3-slot token.

The first token slot should always be a pointer, either alphanumerical or uniquely identifiable, and be the smallest possible length for its format. The second slot should always be the identifier, and no identifier should also be in the hot-swappable arrays (2cd Generated-Pattern array, 3rd Generated-Pattern array, 7th Array, Index Arrays, Mixing Arrays). The first token of the full reference array in both possible cases, or all possible cases (for non-alphabetic token-objects), should be used for L3+ repeating single characters. The only time the second slot is used for pointers is with multi-tagged tokens; in the example, LOWLOWLOWCAP & LOWLOWCAP tokens for the unique edge cases use the second slot for additional pointers to the same array. The second slot is the main expandable slot to allow multiple pointers, in examples: LOW7tCAP, LOW7uCAP, LOW7vCAP, LOWtuvCAP, LOWLOWLOWCAP, & LOWLOWCAP expand the second slot to give additional context without inflating per character (in most cases). All expanded second slot considerations should be done third-to-last in compression but third-to-do during detangling. The optional third slot should always be the same case as the full reference array case. The optional third slot is to inform of repeating needed to get back the full token that was swapped out.

The tokens that are the replaced sections within a string or data-set should be dynamically chosen or based on another section previously in the string. Any changes to these rules should be included in some form within the programming slots. Determiner slots are considered overflows.

When adding non-binary, non-trinary data to the programming or determiner slots, use WaveDataAPIs (https://github.com/DigiMancer3D/WaveData-HAPI).

Setting the Data

We need to make sure there is some order to the chaos so we have an Application Binary Interface (ABI) so we all can have a similar data structure even if we don't all use all the sections of the ABI. Next will be the ABI used for setting up our graphs before token swapping into a faux-hash, which is as follows:

Datagraph {

Graphhash [

previous pointer (base64 |or| base91) |or| radius (base64) if no previous pointer

];

Determiner slot 1 [

anything

];

Determiner slot 2 [

anything

];

Programming slot 1 [

magic number (numerical; decimal preferred)

"x"

micro-loop point (numerical; decimal preferred)

"e" (normal encoding) |or| "f" (compressed encoding) |or| "g" (graphed encoding)

input length (numerical; decimal preferred)

];

Determiner slot 3 [

anything

];

Row 1[

anything,

anything,

anything

];

Determiner slot 4 [

anything

];

Row 2 [

anything,

anything,

anything

];

Programming slot 2 [

index

];

Row 3 [

anything,

anything,

anything

“.”

Row 0 [

anything

]

];

}

Utilizing the example table and information used so far we will transmute "*Hello World*" into a basic SuPosXt (SPX) hash.

Input: **Hello World**

The example input string

Input Length: **11**

The example input length

Input Sum: *(((1.5 * 18) + 11) – 1 =* **37**

*The example input sum (ABSOLUTE((Full_Action_Counter * Max_Loop_length) + Input_Length) – 1)*

Loop Point: **1 [*x18*]**

The example input Loop Position (Round-Up[Previous _BIT + Length / 18]) ← [initial bit or position is always the previous set end bit]

NOTE: **There is not always a Previous_BIT, in this case there is no previous bit**

Micro-Loop Point: **2**

The example input Micro-Loop Position (Length % 9)

Full Action Counter: **1.5**

The example input Full Action Counter ((Loop Point + Micro-Loop Point) / 2)

*NOTE: t***he average sum of the loop point & micro-loop point**

Node Output:
[D1D0]0.2[A1A1A1]111[A0B0C0]6[A0B0C0]6[A1B1D0A1B1]55.01[D0]0[A0B0D1]50.1[A1B1D0A1B1]55.01[A1B1D0A0B1]55.01[A0B0C0]6[A0A1A1]111

The example input vis-a-vis **Node** *(the* **EE2E font***)* **[DNDregular.otf]**

Faux-Binary State:

Data rail A: **1110011011100011**

Data rail B: **001101110**

Data rail C: **000**

Data rail D: **1000100**

Index: **0.21116655.01050.155.0155.016111**

The example input 17-Grid Graph [Listed View]

Full Grid:

1110011011100011;001101110;000;1000100;0.21116655.01050.155.0155.016111;

The example input 17-Grid Graph [String View]

Shifting Data:

Data rail A: **1110011011100011 → 1110101001101101**

Data rail B: **001101110 → 010100110**

Data rail C: **000 → 011**

Data rail D: **1000100 → 1011001**

Index: **0.21116655.01050.155.0155.016111**

The example shifting input 17-Grid Graph [Listed View]

Full Grid:

1110101001101101;010100110;011;1011001;0.21116655.01050.155.0155.016111;

The example shifted input 17-Grid Graph [String View]

Parallelogram:

A|*D:* **0**

B|*C:* **180**

a: **37**

b: **685**

h: **1369**

P: **1444**

Area: **468768**

The example Alignment Check A

Sphere:

Volume: **1342730154**

radius: **684.5**

The example Alignment Check B

Magic Number: **101**

The example Alignment Check

Chain Weight: *(((((11 * 3.14) + 37) % 2) ^ (3.14 + 11))) / 37* = **13**

*The example weight Round-Up(ABSOLUTE(ABSOLUTE(CEILING(TRUNC(Input_Length * PI) + Input_Sum) % 2)^ABSOLUTE(PI + Input_Length))) / Input_Sum)*

Used Pattern-Based Token(s):

SPX Tokens	Array Tokens

Used Quantum Token(s):

Reference Token	QEC Replacement Token
1001	**2Bd**
118	**Vc**
101	**Pc**

The example token(s) used for quantum entanglement

Graphed Hash:
LTY4NC41cg===;1;1;101x2f11;1;1110101001101101;0;010100110;0.21116655.01050.155.0155.016111;011.1011001

Compressed Graphed Hash:
LTY4NC41cg0a18a101x2f118a17x10017w8a010100119aovsvc7Bqu0puquB6sv3A118A1011001

Quantum Compressed Hash:
LTY4NC41cg0a18a101x2f118a17x10017w8a0102Bd19aovsvc7Bqu0puquB6sv3AVcAPc2Bd

Input Uninterrupted Binary: **01001000 01100101 01101100 01101100 01101111 00100000 01010111 01101111 01110010 01101100 01100100**

Total length: **88 (L88)**

Output:
LTY4NC41cg0a18a101x2f118a17x10017w8a0102Bd19aovsvc7Bqu0puquB6sv3AVcAPc2Bd

Total length: **73 (L73)**

Output Uninterrupted Faux-Binary: **11101010 01101101 01010011 0 011 1011001 0.211166 55.01050 .155.015 5.016111**

Total length: **67 (L67)**

Looking at the end results shows that these quantum manipulations do indeed lower the length of the binary equivalent of the inputted string, even after index & parity inflation. The end results may not seem magical but do showcase how we are superpositioning text with its index & parity checks while still reducing the size of the end string. Overall, in testing, the reduction of data does seem to grow in difference the longer the data is from the full reference array. The quantum entanglement method may seem unorthodox to use unassociated single-character or multi-bit tokens, but when we use this alongside the rules put in place, we place tiny amounts of technical data without adding more information. Because of the rules and controlled order, we can use quantum entanglement in multiple ways and even stack the replacement tokens on top of each other in situations that would otherwise be irreversible but stay potentially reversible.

Even if we only use the QEC token swaps, we can still reduce the length of almost any hash or repeating-pattern strings without needing to know much more than it is a quantum manipulated hash string, which can be as simple as front running: QEC |or| Magic Number (+) ”x” (+) Micro-Loop Point (+) e |or| f |or| g (+) Input Length, just as this methodology suggests.

--3Douglas “3D” Pihl

I would like to add a special thank you to Jake La`Doge in assisting in the creation of the self-correcting methods and directing me to the "Futurama Theorem," which ended up being the glue to getting this concept system to work dynamically with verification.

I would like to add another special thank you to The Mota Club (on Telegram) for helping in motivation and spelling corrections for the previous SPX papers and concept.