:: Snerx's N-Dimensional Cipher ::

Matthew R. Garon - 2018/3/10

SNIPH Whitepaper

ABSTRACT: Sniph is a dynamically shifting multi-dimensional cipher. The impulse for creating this cipher was twofold: to create the first scalable N-dimensional cipher and to create the first cipher that is *in principle* impossible to knowingly bruteforce. This is accomplished by using a combination of several traditional cryptographic techniques as well as a new one termed library surfing, with segmentation between the two that makes things like perfect forward secrecy obsolete since the keys can never be reverse-engineered without creating a hermeneutical lacuna, or gap in the knower's capacity to know which keys were used with the original plaintext. Further, I wanted to be able to use a homebrewed cipher without higher-maths so the entire process could be easily done by hand.

:: Introduction and Meta-Epistemic Concerns ::

I want to start by limning exactly what is meant by 'in principle' when I say Sniph is impossible to knowingly bruteforce *in principle*. Epistemically speaking, in principle, nothing is locked away in principle. What is meant by this is that there is no possible information that is locked away from any knower for principled reasons; only for practical reasons could information be locked away, further that this is *necessarily* the case. Wittgenstein famously proved this with his private language arguments which showcase that any principle gap or divide in information between knowers results in an ultimate and total divide between the knower and all possible knowledge.[1] This is to say private language leads to a complete collapse of any epistemic framework you want to posit - it makes knowledge itself impossible.

Many people have a hard time understanding Wittgenstein's private language arguments, and my two-sentence summary may be disagreeable to some, so instead I will give a far simpler original epistemic example that illustrates this point. I like to call this the *opaque box* example. Let's say there is a box somewhere in the universe that is totally opaque, not just to light, but to any possible probing or analysis. We cannot know what is inside this box, and not just for practical reasons like lack of proper tools (similar to what José Medina termed a *hermeneutical lacuna*),[2] but for principled reasons like lack of *any* possible understanding. Okay, so we can't know what's inside the box, and we can't know in principle, it's locked away from all possible knowers. Anything then can be inside this box, and anything means *anything*. This includes information that might make knowledge itself impossible. The box could contain information or conditions that make its own existence, or even potency for existence, impossible. This means the opaque box self-defeats. It is an impossible object, like a square-circle; the thing in itself allows the impossibility of itself and forces its own denial, therefore you cannot have that thing, and necessarily so. This is apodictic, so anyone who argues to the counter can be dismissed as misunderstanding what it is they argue for.

Okay, fine you say, but what does this have to do with cryptography? Well we know that information can only be locked away for practical reasons, which means we now know that *all* ciphers and encryption algorithms work off of *when* not *if*. This is to say that all ciphers must necessarily be concerned with the amount of time it takes to bruteforce them, not if they can be bruteforced in the

[1] Specifically from remarks 243-304 in *Philosophical Investigations* - https://snerx.com/archive/Wittgenstein-PhilosophicalInvestigations.pdf
[2] José Medina, *The Epistemology of Resistance*, Oxford: Oxford University Press, 2012.

first place. Some might point out OTPs as an exception to this, that if each pad is truly one-time, then it would be impossible to decipher the ciphertext because the keys used to establish it are truly unique, or *truly random*. Claude Shannon was the first person to mathematically prove this impossibility, but it should be noted that math is second-order logic, and is subservient to first-order logic. In the same work Shannon illustrates many other important properties of cryptographic systems like problems with symmetric keys concerning the use of the same key to cipher as well as decipher.[3] Keep all this in mind for the latter sections.

If OTPs could actually do this, they would be wildly useful because they would allow for a *principled* lock-away of information, where no amount of probing could ever recover the plaintext. But everyone reading this now knows that principled lock-aways are not possible given the meta-epistemic conditions for knowledge. So OTPs are necessarily stinted in their use, and this partly explains why they haven't caught on: if it's truly random, then you need to somehow communicate a new keyset for every new ciphertext, which becomes very impractical very fast (and also requires automation, stealing control of the keys away from human users).

As a potential solution to this problem, there are ciphers that implement key streaming, which is meant to allow for wrapped or modular addition of characters, as if to make an OTP infinite in its list of keys. This treats the symptom but not the problem itself, because if you know the PRNG source that is used to deterministically generate those keys, then you can break the OTP just the same since the 'password' is effectively the PRNG source. 'True' randomness is possible if you want to use robust CSPRNGs (like Cloudflare's LavaRand),[4] but this is highly impractical for one-off messages, or the freezing of information for documents like encrypted PDF's, or if you don't have an internet connection, or if you want to have direct control and access to the keys being used to encrypt the data (again, automation ruins this).

An additional potential solution is asymmetrical cryptography (public-key cryptography), which does solve some issues, but not the fundamental issue that a private key can be bruteforced given enough computing power and time. Once a private key is bruteforced, all other messages signed with the same key are now completely open. However, asymmetrical cryptography is so computation-intensive that it is *infeasible* to bruteforce even if given millions of years.

This is not good enough for me, and it shouldn't be for you either, not just because there may be sufficient computational devices invented in the future that are orders of magnitude more powerful, like quantum computers (I have seen some mixed debates as to whether these are actually devastating to current encryption algorithms), but because we are looking to cross the divide between the practical and the principle here. The goal is that even with infinite computing power and all possible keys given instantly, you could still not tell if you've cracked the cipher. And this should always be the goal from here on out in cryptography; why bother doing cryptography anymore unless you think there is substantial room to improve in this regard? For that matter, why bother doing anything unless you want to do it better than anyone else ever will? Touch the sun **and** keep your wings.

Sniph is an attempt at collapsing the distinction between principle and practical modalities at all levels, not just in mathematics for cryptographic purposes, but metaphysically speaking as well for meta-epistemic thought experiments. If this is successful in the way I describe, Sniph will serve as an incredibly useful example for meta-epistemic discussion surrounding modalities and general properties of knowers in epistemic systems on top of its use in cryptosystems.

I try to accomplish this by doubling down on the problems with OTPs and symmetric key systems in general; I believe that when stacked a certain way their problems can negate each other (features, not bugs). This leads to what I term 'library surfing' and is based on a similar conceptual

---

[3] Shannon, Claude. "Communication Theory of Secrecy Systems". *Bell System Technical Journal*, 1949. 28 (4): 656–715. doi:10.1002/j.1538-7305.1949.tb00928.x

[4] This is a good read if you weren't previously aware it existed - https://blog.cloudflare.com/randomness-101-lavarand-in-production/

structure to that of the search algorithm for the Library of Babel's website.[5] Part of this may be considered similar, but not the same as, XOR cipher functionality. A significant difference between Sniph and a XOR cipher is that in Sniph you cannot add/subtract/XOR the plaintext, ciphertext, or keys from one to reveal the other. All this together makes Sniph immune to plaintext attacks. All this also gives Sniph the property of perfect secrecy without the need of communicating a new keyset for every message.

These seemingly fantastical statements may be off-putting to cryptographers, for, "Few false ideas have more firmly gripped the minds of so many intelligent people than the one that, if they just tried, they could invent a cipher that no one could break."[6] But this is not a matter of merely gaining a 'sufficiently large keyspace', but of gaining an *infinite* keyspace, a topologically recursive keyspace, an indefinitely traversable keyspace, with no way of knowing if any of the space traversed is meaningful space unless you were the one who assigned the keys in the first place.

So when I say Sniph is impossible to bruteforce *in principle*, I am saying that it is impossible to determine *meaningful keys* qua bruteforce attack. I understand this may be an unfair use of the term 'in principle' here since the relevant keys can in fact be bruteforced, and are expected to be bruteforced, but this technical abduction of the term allows us to speak on the interpretive gap the bruteforcing process creates without having to rely on purely meta-epistemic terminology like the earlier referenced *hermeneutical lacuna*.

An outline of how the protective property of Sniph works - If you obtain a plaintext and paired ciphertext, you will be able to traverse all possible keys within the given space defined by the length of the ciphertext (the length of the ciphertext effectively determines the analogous page-length in the Library of Babel, this is explained in painful detail in the next section). Every single one of those keys will be able to produce *all possible* plaintexts from *all possible* additional ciphertexts, because in traversing this keyspace you will have produced *all possible keys*. This means having plaintexts with paired ciphertexts will not reliably produce the keys you desire. Unintuitively, a more accurately way of saying this same thing - it will *always* produce the correct keys, but they will be mixed in with all other possible keys that all work equally well and so you will never be able to know which keys were the original ones you desired, only that they were definitely produced.

Sniph originally started as a project to create a sufficiently-difficult-to-crack cipher that could be done by hand on paper for the use of sending encoded messages between friends in board games. This means that this is all done without the need for higher maths or any computational machines. Big claims from a dilettante, I know, but hopefully the current public code and following descriptions substantiate these Icarusian claims.

[5] The principle the search function operates on is what I believe the most direct parallels to Sniph can be drawn to - https://libraryofbabel.info/
[6] I am quoting David Kahn, from his book *The Codebreakers*.

:: Process and Properties of the Cipher ::

OVERVIEW: The way Sniph works is by generating tables of variable sizes (like Polybius squares) that are then seated inside of each other to the N*th* dimension specified by the user in the cli. The breadcrumb/pathway through each dimension/table-set is chosen based on a streamed passphrase. This process, with additional steps, is repeated for every single letter of plaintext fed through the cipher giving it similar, but not same, properties of one-time pads. Seating tables of variable sizes gives this cipher the additional property of having one-to-many inputs-to-outputs and one-to-many outputs-to-inputs, but only sometimes many-to-many inputs-to-outputs. The ciphertext generated and this property of it can be thought of as being similar to a dynamic version of the Spanish Strip Cipher. This directional one-to-many property helps significantly with obfuscating data; when this property is stacked, this is what I call library surfing.

IN PRACTICE: You start by selecting an arbitrary plaintext to cipher, like 'HELLO', or 'WORLD', and by selecting an arbitrary passphrase like, 'ALICE', or 'BOB'. After this, the first thing that the cipher needs to do is generate the tables; the cipher decides what table dimensions to use either by flags sent in the CLI, manual entry in the CLI, or by the given passphrase. For this example we'll use the passphrase so we can see how the process works in its *natural state*.

Given the passphrase 'BOB', the cipher procedurally generates the table width, height, and then depth. To do this, we look at the first letter of the passphrase 'B' and take the numerical relation of that letter '2' and use this to decide that the table width will be 2. Then we look at the second letter 'O' and take this to mean the table height will be 15. Following the same procedure, the depth is then 2. Due to the restraints placed on table dimensions by the charset being used (understood later), the minimum table dimensions that can be used are 4x4x4. This means that we will instead be adding each of the numbers we just found to 4. The table dimensions are then 6x19x6. While the cipher itself is N-dimensional, we still want to be able to easily do this by hand on paper, so we will be wrapping any numbers over 10 so that our tables on our graph paper don't become too large. The table dimensions are finally 6x9x6.

Because N-dimensions means *N-dimensions*, you could manually set the table dimensions to be *1,000 x 10,000 x 100,000*, but I obviously do not do that by hand, and anything over 10x10xN is currently not implemented in the code (but will be in the future).

Now we have to populate the tables with characters so that a plaintext character can be chosen. Since these tables have depth, or *seated dimensionality*, and because drawing and filling out a cube on paper is difficult, we will use the table depth (all the stacked 2D layers) up until the last two layers (*N*-2) merely as a means to produce a Caesar shift in our cipher.

To do this, we look at the first table, 6x9x**1**, then 6x9x**2**, then 6x9x**3**, then stop at 6x9x**4**. The first table is 6x9, and to determine which cell to open, we look back at the passphrase. In this particular case, all the character spots of the passphrase have already been used, so we will wrap the passphrase over to give us additional characters (streaming the passphrase into 'BOBBOB'). The passphrase's fourth letter's numerical relation is 2, so we go 2 across from the left of our initial table, giving us the second column. The passphrase's fifth letter's numerical relation is 15, so we go 15 down from the top of the second column, giving us the sixth row. We represent this cell as '2-6'. We repeat this process for the second 6x9 table opened, using the sixth and seventh characters from the passphrase 'BOBBO**BB**OB', giving us '2-2'. The same with the third table, '6-2'. And with a final passphrase of 'BOBBOBBOB**BO**B', the fourth table, '2-6'. These are the *path-tables* and are used to determine the Caesar shift for the final two *end-tables*.

Now that we've determined the path-tables, we will sum each table's picked cell and then add each sum together for a total which will be used to shift the character sets in the end-tables. We start with the first table of '2-6', which gives us -4. The second gives 0, the third 4, and the fourth -4. Adding these all together we get negative four (-4 + 0 + 4 + -4 = -4). You may notice that working with a very short passphrase and lots of repeating numbers gets us nice small results to work with all the way through, but of course this is arbitrary in the code and passphrases with higher entropy result in more dynamic numbers throughout this process.

We can populate the end-tables now. This is visually represented in the original gdoc I made about Sniph. The second-to-last end-table's first cell gives us our first last end-table. I am aware of how poorly named these things are, just bare with me. To reiterate, in this example there is a fifth 6x9 table in the total of six sets of 6x9 tables; this table is our first 'end-table', or our second-to-last table total. Picking any cell in that table opens up another 6x9 table, which is our second 'end-table', or our last table total. Hopefully this clarifies it. The gdoc is there to help if imagining this gets too tricky.

So we can look at the first cell in our second-to-last table here, which would open a final 6x9 table. This first-cell-final-table's first cell would be populated by the first character in our character set, and for this example our character set will be 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. That makes the first cell A. Going across the row to the second cell, we would see B, then C, then D, then E, and finally F, which ends the row. The second row starts with G, and the character set continues to wrap row by row until it runs out. Again, this is visualized in the linked gdoc. In a 6x9 table there are 54 cells, which means at the 27*th* cell, where the character set would otherwise be finished, we wrap the character set again and start at A, then B, and so on. At the 53*rd* cell we would begin wrapping the character set yet again. This means this table's last cell (the 54*th* cell) ends with the character 'B'.

Opening the second cell in our second-to-last table here, we would find another final 6x9 table. This second-cell-final-table's first cell would be populated by the character now wrapped over from the end of the first final table, which in this case is 'C'. The second cell in this final table would be D, and so on. The character set keeps wrapping and continuing over and over again until each final 6x9 table is populated. At least, this is how it would be if we started the first cell of the first final table with the first character of the charset.

Instead, our path-tables told us that the character set was to be shifted back four characters, and so instead our charset looks like '**WXYZ**ABCDEFGHIJKLMNOPQRSTUV'. This means the table-populating process is carried out the same way, but with different starting characters. This is how the Caesar shift is applied in Sniph.

Why split up the end-table into two sets of equally sized tables? Why wrap the charset over and over if you only need one character to be chosen? This property of Sniph, using seated tables, allows us to arbitrarily pick one of many possible outputs to code for our plaintext character. This means that encoding the same plaintext character over and over using the exact same passphrase, the same pathways and table dimensions, will generate many different ciphertexts. This makes the re-use of a symmetric key secure since there are no consistent outputs that can be amalgamated.

Getting back to this in action, if our plaintext that we want to cipher is 'WORLD', and our shifted character set has populated the first final table, we now have three choices of cells to pick that contain the character 'W'. There are two more choices in the second final table, and so on. We can pick any of these cells to represent 'W', the choice is arbitrary. In the code, this is chosen at random. On paper, the choice is at the pen's discretion.

Let's say we picked the very first cell of the first-final-table, this would be cell '1-1' of the second-to-last end-table, and cell '1-1' of the first-final-table. This is then the ciphertext '1111' that codes for the plaintext 'W'. If we had picked a W from the second final table '1-2', at cell '5-1' for example, then the ciphertext would be '1251'. There are of course many other cell locations that code for 'W' in this example. *N.b.*, having many outputs for a single input does not itself make it any more difficult to

reverse this cipher. Rather, the final transformation of the output as ciphertext is what makes this property notable.

If the end-table's coordinates were always direct like this, where the first cell of the first table is always '1111', this would make it obvious as to which cell of which table was selected regardless of the table dimensions. This also means that if a long ciphertext never uses a number above 4, then an adversary intercepting the message would know the table dimensions were 4x4. To stop this kind of analysis, the un-used numbers for a table dimension are wrapped as stand-ins for the already used numbers. In a table-size of 6x9, since the rows are only 6 long, 7 can stand in for 1, 8 for 2, 9 for 3, and 0 for 4. Since the columns are only 9 long, 0 can stand in for 1 there. *E.g.*, our first choice for 'W' as '1111' could also be represented as '7070' or '7110' or some other combination. This is also chosen at random.

To be clear, not only are the cell choices many/arbitrary but the cell *representations* are also many/arbitrary. At least they are for table-sizes less than 10x10. This stops analysis of ciphertext from being able to determine the table sizes being used, since all numbers for all sizes are always present in all ciphertexts. Additionally, the ciphertext only ever tells you the cell locations being chosen for the end-tables, they don't include any information regarding table depth, or the seated dimensionality, so the Caesar shift cannot be determined from analysis of the ciphertext.

This is uncommon for a cipher, since most ciphertexts that have been Caesar-shifted can be directly shifted without damaging the plaintext message, whereas this is not the case with Sniph. Directly shifting the ciphertext output from Sniph ruins the possibility of being able to reverse the ciphertext into plaintext. And now for an even bigger claim, this property of Sniph makes it possible to have both randomness **and** predetermined OTP keys in the same cipher. I will say this again but in a different way - this allows you to use unique OTP keys without having to transmit the OTP keys each time you generate ciphertext and allows you to reuse the same passphrase and not generate similar-looking output each time. This solves a fundamental problem with the usage of one-time pads not being one-time.

What we end up with here is four numbers coding for a single character. We have our four-character long ciphertext for our single-character plaintext. All plaintext ran through Sniph is expanded by a factor of four. The more perspicacious amongst you may have realized that the same principles of Sniph can work for a single table with a Caesar shift applied, and so we would only need to expand text by a factor of two, however I made the original pen-on-paper cipher when I was 13 and didn't think this far ahead. In the future, the code will probably reflect more efficient uses of the cipher in this way, but for now it works the way it works.

That's our first character of plaintext. If we moved to the second character of plaintext now, we would repeat the entire process described up until this point. When doing this by hand on paper, I have been re-using the same table dimensions for all plaintext characters to save time, however in the code for Sniph this is not the case, instead it re-determines table sizes per-character of plaintext. This means that while our first character got tables of the 6x9 size, our second character would get something different (or possibly the same again if the passphrase happened to wrap around starting over on 'BOB'). This repeats for each character being ciphered, and the passphrase is streamed, or continuously wrapped, until all characters have finished being encoded.

BREAKING SNIPH: That was Sniph in a nutshell. To understand this in more detail, and to see why this is so devastating to attackers trying to bruteforce Sniph, we can quickly try to reverse this process.

Let's say we were an attacker, and we knew how Sniph worked, and we intercepted the ciphertext '16671251565263370190' (v0.5 of Sniph output this from the plaintext of 'WORLD'). Okay, we want to be able to determine the plaintext that the ciphertext codes for, so we need to guess the passphrase used to determine the table dimensions. We bruteforce the ciphertext starting with 'AAAAA'

and crawl through every possible permutation, ending with 'ZZZZZ'. Given current standard commercial processing power, this should be instantaneous to bruteforce. The bruteforce program immediately shows the entire list of characters that '16671251565263370190' could code for given each possible passphrase used. One of the passphrases bruteforced was 'BOBBO', which was indeed the original passphrase used (given that it wraps). However, every possible five-letter plaintext was generated in this list given every possible five-letter passphrase. This becomes very problematic very fast.

Something interesting happens here. Yes, you will have generated the correct plaintext after hitting the correct passphrase of 'BOBBO', but you will have generated the correct plaintext with many other passphrases in the possible permutations as well. Many passphrases will get you the desired plaintext, however only one will get you the desired plaintext in the desired table dimensions. There are a couple notable things going on here, to start - this is the second time the one-to-many property is functioning in Sniph. The first was in selecting the coordinates for the ciphertext, the second here is in selecting passphrases to match ciphertexts. The second notable thing going on here is that the table dimensions do not have to be determined by the passphrase, the CLI user or paper user can arbitrarily select their own table dimensions, making passphrase bruteforcing an order of magnitude more difficult.

So this time, in order to find the passphrase used with an independently chosen table dimensionality, we will have to try every passphrase permutation (from 'AAAAA' to 'ZZZZZ') for every possible dimensionality. But that's fine we say, because we can still spit out all the possible permutations instantly. So we run the bruteforce program again and get a list of plaintexts generated by passphrase permutations 'AAAAA' to 'ZZZZZ' for the 4x4x4 table dimension. We get another list for the 4x4x5 table dimension, and another for the 4x4x6 table dimension. Wait, how many seated dimensions do we try?

Well, with a 26-character-long charset, the total number of Caesar shifts that would be meaningful is 26 shifts, since the charset becomes mirrored for each table permutation after 26 shifts. So we say we'll go out to NxNx26 tables, since meaningful shifts are not likely to happen beyond 26 transformations. Okay so we start over with 4x4x4 to 4x4x26 and then repeat the process with 4x5x4 to 4x5x26 again, all the way to 10x10x26 since we're working with base-10 numbers in the output. I'm sure someone smarter than me will be able to say how many total sets this amounts to.

Let's pretend this would still be instantaneous, and we get our list of plaintexts paired with each possible passphrase in each dimension-set for our single ciphertext. The problem now is compounded, for by allowing *every possible dimensionality* to be explored, with *every possible passphrase*, we will have generated *every possible plaintext* in a five-character space. This is the third one-to-many property of Sniph.

This third one-to-many property of Sniph works on every single pairing of ciphertext and passphrase; you can generate every possible character-combination as plaintext. This means that in our example of a twenty-character ciphertext, we will have generated every five-character latin-script text that has ever existed, will ever exist, and can ever exist. Every possible permutation within a five-character space will have been covered, and further, it will have been covered many times over. This means that not only will we have many valid English plaintexts that can stand in for our ciphertext, but many valid N-language plaintexts that can stand in for our ciphertext, all possible valid plaintexts actually, and multiple sets of them. This means there is a recursive multiplicity to our generated sets. Additionally, there will be many passphrases besides 'BOBBO' that work to generate 'WORLD' in the same and different dimensionalities given while bruteforcing.

Our last attempt at breaking Sniph, which is the most interesting - this time we're not blindly bruteforcing a ciphertext from Sniph, we intercepted the original plaintext as well and are trying to determine the passphrase used so we can decipher future transmissions. As the attacker we perform this plaintext attack; we know the plaintext was 'WORLD', and we bruteforced the ciphertext just the

same as in the last attempt. This time we only care about passphrases that result in 'WORLD' given the ciphertext, and this leads to the fourth and final one-to-many property of Sniph.

All passphrases result in 'WORLD' given the ciphertext. Since the dimensionality of the tables can be determined independently of the passphrase, there will always be a possible dimensionality wherein any given passphrase can generate any plaintext from any ciphertext. You can find a valid combination of plaintext, passphrase, dimensions, and ciphertext, for any given plaintext, passphrase, dimensions, or ciphertext. This is what it means to surf the library.

To recap this section: You start with some arbitrary plaintext, like 'HELLO' or 'WORLD', which gets you some arbitrary ciphertext '11112222333344445555', and you try every password permutation from 'AAAAA' to 'ZZZZZ', which gets you every plaintext permutation from 'AAAAA' to 'ZZZZZ' in every dimensionality. This means a robust plaintext attack will result in many (read: all) valid passwords in that five-character space.

I want to briefly expand on this particular aspect here; that five-character space is what I am calling a *library*. This is because for each set character-space, there is every possible character permutation that can be crawled through, very similar to the property that the Library of Babel has. Although, the Library of Babel is only one library, its algorithm only crawls through a 2,500-character space. Sniph has infinitely many possible libraries, since any length can be used as a plaintext input for ciphering (or a ciphertext input for deciphering, divided by four). Note that computational difficulty for bruteforcing Sniph is not based on how long the passphrase is, bruteforcing is exponentially more difficult the longer the ciphertext is.

Library *surfing* is not used to reference the process of ciphering plaintext in Sniph, but rather to the process of deciphering ciphertext. This is because you are surfing between libraries, between possible sets of permutations, when you attempt to decipher in Sniph. This is also the source of the talk much earlier about 'hermeneutical lacunas' since there is a gap created when attempting to interpret the output from a bruteforce attack.

So one-off plaintext attacks in Sniph do not produce reliable information. Further, all passwords that fail in one dimensionality work in some other dimensionalities, so all passwords always work. All that said, there is a potential way to break Sniph given sustained plaintext attacks; this is covered in the next section.

:: Remaining Issues and Potential Attacks ::

There is one potential issue that can be exploited for a cipher-breaking attack in Sniph that I am aware of. Regarding plaintext attacks, a single paired plaintext and ciphertext cannot meaningfully produce the correct keys. However, if multiple plaintexts are given with their matching ciphertexts, I believe there may only be *some* keysets that match up in the same dimensionalities between the pairs. The more ciphertexts you test will narrow down, very swiftly, which of those keysets is valid because very few, if more than one at all, will work with the same dimensionality each time.

If a hard segmentation exists between plaintext, ciphertext, and passphrase/keyset, it would be impossible to determine one from the other no matter how many samples you obtain. However, I am not competent enough to test this. So until someone who is, does, I will put on my dilettante-engineer hat and try to ensure triple redundancy of the already-existing property. Triple redundancy will at least make it certain that this issue is as I described.

The code will be augmented soon so that additional to each plaintext character getting its own Caesar shift and table dimensions, it will also get its own charset size. The Caesar shift and table dimensions are easy to understand; each character has a shift transformation and table dimensions generated for it as NxNxN. The charset size is different; each plaintext character will get its own charset generated for it as well, with the minimum size being 96 (since 96 base characters in the code need to be represented), but *N* additional characters repeated at the end. This is not the same as shifting the characters, e.g. 3 additional characters means that the end of a character set will wrap as, "X, Y, Z, A, B, C, A, B, C, D, E, F..." and so on. This creates a stutter in the charset.

Since they could all be determined from the same passphrase, and the process for their generation will be known, if there is a truly devastating flaw with library surfing, then it will be exposed even after this additional modicum of entropy is added.

I am not a cryptographer, and while cryptography is fascinating to me, I am not good at it. I gave and received a lot of opprobrium about my cipher during its initial conceptual incubation on lainchan.org because I was claiming to have figured out something substantial that no one who knows better has figured out. However, I am a trained logician and have considerable background in meta-epistemology; this is how you can make the claim that you are able to determine some advanced properties of cryptographic systems without knowing anything about them prior. But I am still stupid and have probably missed something obvious and important and devastating to the whole project. Because of this, I am open to edits and revisions of this document should bastardization of terminology be found, or the cipher itself be shown to be insecure.

I am anticipating someone will find a way to break this cipher, but I stand by my statements in the introduction: why bother doing cryptography anymore unless you think there is substantial room to improve? Why bother doing anything unless you want to be the best at it, or at least solve a fundamental problem that has plagued everyone else up until that point? I hope that in discovering further problems with Sniph, potential solutions to those problems will also be discovered.

If you hated reading this whole thing and think I'm terrible, yet made it all the way to this last page, feel free to yell at me here - snax@snerx.com