Simplexity Cypher – A Casual Treatise

By William H. Donnelly (ca November 2013)

Contact E-mail: Simplexity @ Donnelly-House · net
www.SimplexityCypher.com

• Introduction

Based on my reading, study, research, arguably somewhat minimal knowledge and understanding of cryptology and cryptography, and my lay-person interest in the field, this new cypher is, as far as I know and can tell, a unique method of encryption heretofore unknown and undeveloped. If it turns out it is not unique, it is at least original on my part. I have named it the ***Simplexity Cypher***.

Note that ***Simplexity*** could / should (probably) be considered something of an "Old World" cypher, rather than a more modern cipher used for fast Internet or other transaction processing like DES, AES and others. (that is, used to encrypt "secret coded messages" if you will, and the like, including, perhaps, 'personal datasets' one might want to encrypt for security reasons — however, ***Simplexity*** *might* be usable as a more modern type of cipher)

Although I have been thinking about creating a cypher like this for over 40 years (since ca 1976), I had the original idea for this realized version sometime around 2005. This 'finalized' version is only different from my original idea in that it is more well-thought out and polished, and for the most part completed. But it is basically as I envisioned it then. I have thought about it occasionally over the years, developing it over time, only now making it a concrete reality. This incarnation is fairly complete as it is. Although, it can be expanded upon through various variations, some of which I mention here. ***Simplexity*** can possibly be best described as a "Recoverable / Reversible Data Mixing Cypher".

The intent is to create a "simple cypher" that is easy to implement programmatically, and algorithmically, that is relatively quick in encryption and decryption, and is sufficiently complex overall in its final encryption, so that it is strong, robust, and secure, and literally unbreakable, due to the amount of processing that would be required to do so.

This stemmed and stems in part from my original desire back in the day of using much slower computers ca 1976 (my teen and high school era) and not having a robust (or much of any) real experience with and knowledge of advanced mathematics used in modern-day cyphers, which I found super-complex and confusing (at the time and still do for the most part), as one generally would without that experience and training in advanced mathematics, cryptology, and related subject areas. (DES, AES, etc…)

I basically wanted a very simple cypher that is easy to understand and implement, but will be valid and usable. Even if it was and is in an "Old World" sense. I think I have achieved those ends successfully. This cypher idea spans the world of older hand-implemented cyphers, which would basically be impossible for ***Simplexity***, and the newer, 'modern', advanced, hyper-complex ciphers.

A brute force attack would take hundreds of years (or better) using multiple super-computers to process the data and/or algorithm. (even for smaller dataset sizes)

This is "exacerbated" by the noted, optional, possible variations available (below; and others), which would have to all be taken into account and tested for in a brute force attack.

The encoding and decoding possibilities and requirements are **well-**in excess of yotta-yotta ($10^{24} \cdot 10^{24}$) permutations (possibly yotta-yotta-yotta; maybe even yotta-yotta-yotta-yotta (or better), taking into account all the possible cypher variations). Therefore, even at many multiple petaflop/second calculations (currently, as of this writing, about 36 – and used as a general processing speed, since the calculations used here are not FLOPs), it would easily take at least many years, if not hundreds of years, or longer, to brute force break an encrypted dataset.

The above claims are based on a key of 20 to 40(+) 'characters' with at least 64 possible choices for each key two-character group (as described below = $8 \cdot 8$), or 64^N, where N is the key length and at least 20, but ideally > 30. A *yotta* is only 10^{24} (one trillion times less than 64^{20}) Thereby the yotta-yotta suggestion. (or yotta-yotta-yotta, etc.) I believe my calculations are correct. I know someone will correct me if I am mistaken.

The hundreds of years required for a brute force attack claim is from (for example):

64^{20} iterations (for a 20-character key) = 1.3292279957849158729038070602803e+36 (~1,329,227,995,784 (over 1 trillion) yotta-iterations) Some number of iterations less than this would be required for the brute force attack (at least ^N-1 iterations?). Using this number as the example, divided by ~36 petaflops (floating point operations per second – super-computer processing speed – peta = quadrillion) = ($36 * 10^{15}$) = 36,000,000,000,000,000 = ~36,922,999,882,914,329,803 seconds, then dividing out further time segments, / 60 seconds per minute / 60 minutes per hour / 24 hours per day / 365 days per year = ~1,170,820,645,704 years. (over 1 trillion) So even if the processing speed was greater (because of the simple non-floating point calculations/instructions), and taking into account other required processing, and if multiple super-computers were used, it would easily take hundreds, if not thousands, of years of processing to perform a brute force crack of an encrypted dataset.

It is unknown (to me) whether a faster, more elegant decoding attack would be possible, but the design of the cypher suggests that would not be possible on the face of it — this may, or may not, be provable, one way or the other (or inevitably knowable as true or false). On the one hand, it would be interesting to see that happen, but, on the other, I inevitably hope that the cypher is secure enough that my comments above are true, in part or in whole, or to some minimally viable extent. Ideally, to the extent that the cypher is, at the very least, "secure enough" (if not more so).

Note that, if some of the mentioned variations below are used, the brute force attack may be (effectively, or actually) "infinitely complex". For example, if the starting location offset into the dataset for the processing is not always a known value (e.g. zero; or N constant), then, depending on some of the computing and processing

variations noted, all offset locations would have to be processed for the size of the dataset, as well as the other requirements/variations. So a large dataset (byte-size) would multiply the required processing operations and time to process. This is particularly true when taking into account seed values, etc. It quickly becomes a very large problem to solve. Therefore, calling it "infinitely complex", if not absolutely true, is probably a good description. Whether this would apply to more elegant breaking/cracking attacks is a good question.

• Algorithm Design and Description

The encryption and decryption process is simply a set of operations executed on varying operand sizes across the dataset. Decryption is the "opposite" of encryption. (operator reversal) The operators and operand types are chosen from a processed alphabetic cypher key. (such as an easy-to-remember sentence, or group of words — this method was chosen and used here due to the "Old World" idea, usage, and design — however, the key could be created In any number of manners) The dataset can be any binary 'byte data'. (or a 'byte-oriented equivalent')

The ***Simplexity Cypher*** is actually more of a *Cypher System*, since there are optional and potential variations and additions to the cypher to make it more complex and (potentially) more secure. (some known and mentioned here, and others not mentioned or thought about yet, but applicable and usable) The fact that there are so many potential variations makes the cypher more complex, and possibly or probably more secure, since a "code breaker" would have to perform so many algorithmic processes on the dataset in the attempt to break it. That is, it is most likely "Infinitely Complex" (by design).

Because of the way the cypher works, the resultant encrypted data set is the same data size as the source data. If a smaller size is desirable, the encrypted file could be compressed using any number of compression programs and algorithms. Some compressions would be better (smaller resultant sizes) than others, depending on various factors.

> An aside: I haven't actually looked at the possibility of using this system to create a file from an already-compressed file, made maximally small (minimally small?), that is then compressed again, possibly with a smaller size than the original. I'm just thinking off of the top of my head. That may not be possible, but it's an interesting idea. That is, what I'm thinking is: converting a compressed file into a format that is more compressible, like a pseudo-"text file" of some sort (text files tending to be very compressible) as opposed to a 'compressed binary file'. It would be cool if that worked, but there are probably laws of the universe and mathematics and computing that prevent it from occurring. And there are other issues, like the 'law' of increasingly diminishing returns, etc. That sort of thing starts getting very complex very fast. But sometimes it's fun to think about.
>
> However, that's actually not a bad idea, in and of itself, and probably separate from this project. If you can come up with a reversible transformation algorithm that converts "very raw" (highly compressed) binary datasets into formats that are then further compressible, you could achieve even further gains in compression and smaller size. (maybe) That may or may not be doable, although the gains, if any, might be minimal and not worth the trouble.

• Operators: (0 – 7) – {Oper}

```
0  ADD            next key value  (1, 2, 3, or 4 bytes after OpTyp; key wrap-around)      ; A
1  COMPLEMENT     one's complement (binary bitwise NOT; bit invert)                       ; C
2  SUBTRACT       next key value  (1, 2, 3, or 4 bytes after OpTyp; key wrap-around)      ; S
3  NEGATE         two's complement (arithmetic negative)                                  ; N
4  XOR            next key value  (1, 2, 3, or 4 bytes after OpTyp; key wrap-around)      ; X
5  ROTATE LEFT    next key value after OpTyp LSBs (2=1-3, 3=1-7, 4=1-15, 8=1-31; 0 = 1)   ; L
6  SWAP           (Dits, Nybbles, Bytes, Words)                                           ; W
7  ROTATE RIGHT   next key value after OpTyp LSBs (2=1-3, 3=1-7, 4=1-15, 8=1-31; 0 = 1)   ; R
```

The operators list can be changed, in operation, and/or sequence order. But care should be taken that opposite action operators are not next to each other, such as ROTATE LEFT following ROTATE RIGHT, etc. (especially for non-cyclic Operand Types variations) Note that this also includes the first and last operands. (at minimum for "safety's sake") These operators were chosen because they are all "reversible" and simple, easy and fast to

execute. At least two others could be used, additionally, or in place of two existing operations: INCREMENT and DECREMENT. These are probably the most obvious. I chose to not include these two in my 'default' list because they only slightly change the data compared to the others. But the more operations, the more complex the brute force attack decoding requirements would be in orders of magnitude (?).

• Operand Types: (0 – 7) – {OpTyp}

```
0  Unsigned Nybble Hi      (4 bits)    ; H
1  Unsigned Nybble Lo      (4 bits)    ; L
2  Both Unsigned Nybbles   (4 bits)    ; N
3  Unsigned Byte           (8 bits)    ; B
4  Unsigned Word           (16 bits)   ; U
5  Signed Word             (16 bits)   ; W
6  Unsigned Long           (32 bits)   ; L
7  Signed Long             (32 bits)   ; S
```

The operand types list sequence can also be changed, either in sequence, or the types of operands and sizes. (variation) Implemented software may be hard-coded, or allow the user to change the sequence.

• Cypher Key

The cypher key (encypher and decypher) is meant to be an easy-to-remember (or know) alphabetical sentence using the American English alphabet letters A through Z and the SPACE character. All other characters are ignored and skipped. (numerals, punctuation, etc...)

As noted above, the intent here is to use something like an easy-to-remember sentence, or group of words. This method was chosen and used here due to the "Old World" idea, usage, and design of the cypher. However, the key could be created In any number of manners, as one can imagine and develop, including randomized, not alphabet-oriented, etc. Just do not forget the encypher / decypher key, of course. (or the encrypted dataset is lost forever)

Again, although possibly or probably obvious, just to be clear, considering the "Old World" usage, one could use a memorable phrase and tell the receiver of the encrypted dataset vocally, or pre-agreed upon, or use something like "The second sentence of the third paragraph of Chapter 1 of "Moby Dick" (page 9), English version, independently published paperback, December 30, 2020, ISBN 979-8588578417." (etc....)

A significantly long key should be chosen, at least 20 to 30 characters in length, or longer. The longer the better. (?) 40 to 50 is suggested for "super-encryption". The larger the key, the better the encryption and the more secure it will be. (?)

The Key is converted to uppercase alpha and must not begin or end with SPACE characters. (trimmed) Internal successive SPACE characters can be allowed, or disallowed (compressed – suggested).

SPACE is zero, and A through Z (case-insensitive) are in ordinal sequence from there, giving individual KEY VALUES of 0 – 26. Additionally, the 8-bit ASCII byte values of the key characters are used for KEY DATA VALUES, as well. (see below)

These key values are taken two at a time to select (modulo 8) one of the eight Operators and eight Operand Types as described above. This gives a possible 64 combinations per key character, since the key characters are taken two at a time, but the first of each two-character group starts incrementally from character location to

character location. (i.e., 1 & 2, 2 & 3, 3 & 4, etc..., with the second character of the last key character group wrapping around to the first key character)

Similarly, other alphabets may be used, and/or other character encodings, or other methods. (variations)

The key may also be traversed front to back ('default'), or back to front. (variation)

• Data to Encrypt

The dataset to encrypt should be a significantly large size to better-ensure that hacking the encryption is more difficult (due to processing requirements, and analysis). If possible, the dataset should be padded with other, preferably similar (?), or possibly even random, data. And, the actual encoded data can be located anywhere within the dataset, as long as both users (sender and receiver) agree and know where. As mentioned above, the dataset can be any binary 8-bit 'byte data'. (that is, not just plain text)

The data may be accessed as byte-data for simplicity and speed of operation. ('default') This would only require that 4-bit nybbles be parsed from the byte data, and larger values be aggregated from succeeding bytes.

As a variation, the data may also be accessed as nybble-data, which would take more processing to parse out the nybbles and build the aggregate larger values from them. This will result in slower cryption (I know, 'not a real word', but it is now), especially for 'odd address' nybble locations within the dataset. However, with the speed of today's computers, it may not be noticeable.

• Seed-Modifiers

There are two 8-bit unsigned additive seed-modifier values, one for the Operators and Operand Types, and one for the data values. (or Op seeds may be split into two separate seed-modifier values; so three (or more))

They start at zero, or some predefined value – hardcoded constants, or taken from the key, etc. (variations)

As byte-values are accessed from the key, they are added to the seed-modifier to get the value to use, and that value is stored as the seed-modifier for the next usage.

• Initial Offsets

Initial offsets may be chosen, otherwise they default to zero. (variations) There can be zero, one, two, or three different offsets.

The first possible offset (or ZERO) is the offset into the dataset to start processing. This can be a static value, or incremented or decremented after each process-pass, or a known, chosen value may be added to it or subtracted from it. (the latter chosen carefully? – perhaps one of the changing seed-modifier values described above, or its own; optional variations) The offset is modulo the length of the dataset for wrap-around access. This offset may be an 8-bit byte, 16-bit word, or 32-bit long value, all unsigned.

(care should be taken that additive/subtractive values, if used, are non-zero and also such that they cause the traversal of the dataset fully across all data items — that is, for example, that a situation where only the even data items, or every 3rd data item, etc., are processed does not occur — all data items across the dataset should be acted upon relatively similarly)

The Second and Third offsets are into the Operators list and/or the Operand Types list (one or two values) for initial selection. (modulo 8) These values can be as described above for the data offset value. (optional variations)

• Encryption and Decryption Processing

• Key Operations Processing List Creation

First, the key is used to create the Operations Processing List to be executed on the dataset.

As noted above (the variations), each (successive, next) key-character-value is converted to its Key Value (modulo 27), and the Key Data Value is also kept (ASCII ordinal value, etc.). The key values are used in two-character groups, the first to select the (initial) Operator and the second to choose the (initial) Operand Type. (noting seed-modifiers, offsets, etc.)

If the end of the key is reached, additional values are retrieved by wrapping around to the other end of the key. (front or back, depending on key traversal direction; variation)

If additional data values are required, they are gathered from successive key character values. Nybble and Byte values use the next key character data value, and larger values use additional key character data values to build aggregate sized values. (LSB or MSB (variation); noting seed-modifiers, etc.)

Optional key data value variations:

```
The high order bit can alternately be set and reset.
   (before modified by seed arithmetic?)
The value can be alternately bit-inverted.
Other possibilities exist.
```

The Operations Processing List is built in this manner, incrementing to the next successive key character (that is, the two-character key values are 1 & 2, 2 & 3, 3 & 4, etc. – or some other chosen/decided/agreed upon sequencing; variation), until the end of the key proper is reached. (the last key character will always use the first key character as the second value (or the next key character value if some other sequencing is used)) The length / size of the final list is the length / size of the key (or variation). (always excluding non-key characters, if any (numerals, punctuation, etc.))

List item meta data is as follows: (values and usage, and additional items, may depend on variations)

```
Operator
Operand Type (initial selector; variation)
Operand Value (actual, or Unsigned Long for cyclic OpType variation)
Dataset Starting Offset
Dataset Processing Direction
   (hardcoded forward or backward, or alternating; optional variation)
```

• Data Processing

For encryption, the Operations Processing List is traversed forward (or backward) on the unencrypted dataset.

For decryption, the Operations Processing List is traversed backward (or forward, but the reverse of the encryption direction) on the encrypted dataset. (taking into account the needed "operator reversals")

(a variation is to reverse the starting position (top/bottom) and traversal direction of the OPL)

The list of Operands are all "reversible" in operation, as follows:

```
Encrypt         Decrypt
-------------   -------------
Add             Subtract
Complement      Complement
Subtract        Add
Negate          Negate
Xor             Xor
Rotate Left     Rotate Right
Swap            Swap
Rotate Right    Rotate Left
```

If a non-byte (and non-nybble) operand type is next for processing, and the dataset is not evenly divisible by the size of the operand values, then final values are reverted to the previous smaller size. There are three instances where this may occur:

```
Operand size is a 32-bit long word and there are two or three bytes left to process.
   In this case, the next operation would revert to a 16-bit word operation, and then
   a final 8-bit byte operation if needed.

Operand size is a 16-bit word and there is only one byte left to process.
   In this case, the final operation would revert to an 8-bit byte operation.
```

As each dataset value is processed, an incremented (or decremented) cyclic counter (modulo 8) is used to select the next Operation in sequence.

For example, given a chosen implemented algorithm based on the variations as noted above:

```
The first operation on the first dataset value and operand would be ADD.
The second operation on the second dataset value would be COMPLEMENT.
The third operation on the third dataset value and operand would be SUBTRACT.
Etc...
```

Additionally, another cyclic counter (modulo 8) could be used to select the Operand type, as well, making the process even more complex with variations, while keeping the processing (relatively) simple. (this is a strongly suggested variation to the point where it should probably be a 'default')

• Final 'Analysis'

As can be seen by the algorithm described above, particularly with all of its possible variations, the ***Simplexity Cypher*** is both simple and complex — simple in design, implementation, and operation, and complex in its final encryption strength. (as to brute force hacking attacks; and possibly others) And it is relatively fast in execution. It is (obviously?) NOT meant to be a real-time cryptographic solution like might be used for passwords, or network traffic, or banking transactions, and related uses. My original idea was meant to be something that is very simple to understand, implement and use with older computers (circa 1970s), with relatively fast encryption and decryption, by an individual or group who wanted to pass text messages back and forth or something similar to that. Or perhaps even to encrypt personal data. And to do so without using "modern" cryptographic techniques, algorithms and processes, which the Author-Creator finds hard to understand. (note that, after reading my description of ***Simplexity*** above after a few years of not really thinking about it, the description of the algorithm-process seems more complex than ***Simplexity*** itself, which one can only hope is not due to the writing)

• Cautions, Potential Problems and Issues, Thoughts, Questions, Etc.

1) Although the variations described above are probably more than sufficient, there are other expanded and extended variation extensions that could be implemented, as well. (on fancy or whim, or possibly to actually enhance efficacy) The aggregate data size of nybbles, bytes, words, and longs, even if they are shift-accessed across all 'local boundaries', could be expanded and extended to larger and different areas of the dataset, in part, or in whole. This depends on the variation implemented.

That is, currently, as described above, through multiple process executions across the dataset, the aggregate data items 'morph' between and across all of the various data sizes across each 'local section' of data. For example, across the first several bytes of the dataset, they might be accessed as byte, word, and long, and at another time, long, word, and byte, etc... Other reversible algorithms could also be employed across various larger sections and separated sections, as well, and other similar processing methods, as one can imagine.

One example might be choosing two starting locations within the dataset (based on the cypher key info), absolutely or relatively distanced, and working with those sequentially as two-operand operations, or whatever methods can be employed. However, as mentioned, that may not be necessary, and may not make the encryption "better", more secure, etc. But the various variations are probably extensive and both creatively and algorithmically interesting.

2) Care should be taken to not create a palindromic cypher key, or one that is palindromic in nature, either truly, or in the algorithmic and arithmetic senses. That is, a key that encrypts and then decrypts the dataset, giving a final result of the original dataset, in whole, or in part, and/or an encryption that is of lesser strength, robustness, and/or security. This is more of a (potential) problem for certain variations of the ***Simplexity Cypher*** than others.

3) Depending on the cypher variation used, it is possible that the final encrypted data could be in a state where, even if the cypher key was large (30–40+ characters), the data may be in a state where it could be decrypted in 1, or 2, or 5, or 10, etc., processing passes. This is less possible, and/or more improbable, and possibly impossible, depending on the cypher variations used. Some more simple variations would be more susceptible to these

states. Additionally, the data could be left in a state where it is more easily decoded, by brute force or more elegant attacks. This is probably also dependent upon the cypher variations used.

4) It may be possible to create an evaluation algorithm that can decide if the encryption (and/or cypher key) is appropriately encrypted and how secure it is (or may be).

5) One interesting thought is, what happens if you encrypt an encrypted dataset? (with the same key or another key) Does it become "more encrypted"? Would it be more secure? Would it be more difficult to break, brute force or otherwise? (and, of course, you have the various variations in encryption methods, as well)

6) Another interesting thought is, does this cypher create a sufficiently "random-looking" result? That is, if you encrypt a ZIP archive file, or a PDF document file, or a JPG image file, or a plain text ASCII file, or a binary file versus a text file, is there any way to tell from the encrypted file what the source file was? (clues like this might help with breaking the encrypted file)

I will leave it to others who are more advanced in mathematics, cryptology, cryptography, and cryptanalysis to assess, if they can, how robust the cypher is, or its various variations are, and how strong and reliable it is, and cryptographically secure, etc.

• The Naming of the Shrew

Because ***Simplexity*** can probably better be described as a *Cypher System*, mostly due to its many potential variations, there comes the question as to how to specify, designate, and/or name any one variation. That is problematic. However, here is a suggested naming scheme that might work. It is not complete, but gives a good idea of how one might accomplish such a feat.

The general idea is to suffix the name with appropriate property designations that the cypher variation uses. For example, the provided example cypher mentioned below might be named thusly:

Simplexity:Key=ASCII: A-Z;Oper=ACSNXLWR;OpTyp=HLNBUWLS;KeyDir=F;DatDir=F;OperSeed=13; OpTypSeed=437;DataOffSeed=35976;DataOff=27;

Depending on the implementation and options and features supported, KeyDir values would be 'F' for Forward and 'B' for Backward. DatDir values would be the same as KeyDir, with the added optional suffix 'A' signifying Alternating directions for each data processing pass, if that option is implemented, so 'FA' or 'BA'. Other property values would be used as needed depending on the variation implemented.

Such specificity is somewhat cumbersome, but is technically correct and minimally workable and usable. Additionally, I suppose if certain variations become popular, or known to be more secure, etc., then shorthand nicknames might end up being used. For snicks and giggles, the example variation below will be known as the ***Beelzebub variation***, designated as ***Simplexity Cypher (Beelzebub variation)*** or ***Simplexity:Beelzebub***.

• Implementation, Programming and Use

A ***Proof of Concept Example Implementation*** of a variation of the cypher, known as the Beelzebub variation, is available for testing and edification on the following web page.

http://www.donnelly-house.net/simplexity/simplexityexample.php

It is written in the PHP scripting language, and therefore is relatively slow in operation, but works. Some of the source code is available for educational purposes.

• Code Breaking Cracking Challenge 2021

Below is a link to a ~47K encrypted file that was created using a variation of the ***Simplexity Cypher***.

(it is not the Beelzebub variation; not that that may matter; that is, it may or may not make it easier to break if the specific variation is known, although it could help, depending on the variation, especially if the implemented version was not as robust for some reason(s))

There is also a ZIP version of the file created using 7-Zip. Interestingly, the ZIP archive is 125 bytes *larger* than the encrypted file. Whereas the unencrypted source file ZIPs smaller than 47K. (not to give away too many clues, if that even makes a difference) In this instance, one might want to compress the source file and then encrypt that archive file.

A stray thought I had, that may or may not have some validity to it, is that because the encrypted file is in such a state that it cannot be compressed smaller than it already is, that might suggest that the cypher is a good concept and is creating nicely-encrypted results. It could obviously mean nothing of the sort. But it seems like "that (probably) means something". What exactly it means, if anything, is beyond me, but, if I'm 'lucky' it means something good. Of course, that all remains to be seen.

The award offered to anyone who can break / crack this example code dataset has been increased from the original US$100 (one hundred dollars U.S.), and subsequent US$1,000 (one thousand dollars U.S.) as of December 2017, to US$5,000 (five thousand dollars U.S.) as of March 2021.

The award amount may change (increase) over time as an additional incentive. (or, if someone would like to be the Award Benefactor, that would be nice, especially if the amount was high enough to garner increased interest — I have actually, 'effectively', reached this point) Note that this higher amount, if awarded, **will** need to be paid off over many years (probably over 5 or so years now for $5,000), but it WILL eventually be paid, if won.

Also note that the code breaker(s) (individual or team) needs to "show their work", have it be understandable and reproducible, and allow it to be fully published on this website (though not necessarily first-publication if they wish to publish elsewhere first). Other rules and regulations may apply, at my discretion, but I will endeavor to be fair about it all.

Use your right-click mouse button context menu to "Save link as..." (or whatever equivalent your browser uses) to download the **CrackMePlease.simplexity** encrypted file. This is a binary file so it is not viewable in your web browser or text editing program.

http://www.SimplexityCypher.com/CrackMePlease.simplexity

http://www.SimplexityCypher.com/CrackMePlease.zip

Good luck. Although I obviously hope that the encryption is unbreakable, it would be cool if someone actually figured out a way to do it. But it would be cooler verging on awesome if it was "impossible".

If, at some unknown point in the future, no one is able to break it, I will publish all pertinent information about the variation used, the key, the "proof of concept" implementation that was used to encrypt, and needed to decrypt, the source code, etc...

For the record, the approximately 47K file encrypts and decrypts in about 45 seconds using the (slow) program I used. I might write a faster program to see what some real-world executable times would be for encryption and decryption (the timing for which should be very close, I think). My programming language of choice will probably be some compilable version of BASIC. My guess is that they would be fairly fast, relatively speaking. I would think this file would encrypt and decrypt in well-under a second.

• **Comments**

Comments, input, and constructive critique are appreciated.

See the comments page to post a comment or read comments.

A contact e-mail address is provided at the top of this document.

This page is available as an ~170KB PDF document.

Last Modified: Monday, March 1, 2021 – 5:00 pm PST/PDT

Previously Modified: Thursday, December 7, 2017 – 2:00 am PST/PDT

Previously Modified: Friday, June 13, 2014 – 8:00 pm PST/PDT