Automated Cryptogram Solving, Volume I

Design Guide to Automatically Solve Almost All of the ACA Cipher Systems by Computer

By SHMOO

Volume I covers: Preface, Introduction (vital to the rest), Simple Substitution, Checkerboard, Monome-Dinome, Simple Substitution Variants, Null, Baconian, Keyphrase, Tridigital, Rail Fence, Redefence, Route Transposition

Preface

Welcome to “Automated Cryptogram Solving”. This is meant to get a typical member of the American Cryptogram Association (ACA) started on writing computer programs to *solve* cryptograms.

Who is this for? Someone who likes using computers to *automatically* solve historical ciphers. Someone with intermediate coding skill.

Who should stop reading right now? People who only want to solve such ciphers by hand.

What if the reader just wants to learn how to do frequency counts and or interactive solvers? See the companion volume, *Elementary Cryptographic Programming*.

Elementary Cryptographic Programming would also be a useful departure point if one is newer to programming generally. In that case, master the “elementary” booklet first and then come back here.

An important underlying assumption: This work assumes the reader has access to *The ACA and You.* This book (of course) calls out cipher systems by name. But, the mechanics of enciphering and deciphering are amply described elsewhere; not here. If you want a comprehensive list of these systems, obtain The Cryptogram Index, currently at https://www.cryptogram.org/members/downloads/CmIndexND2023.pdf and then find the index entries for *Novice Notes.* Or, download the *Practical Cryptanalysis* series from the same website (five volumes). Either covers most ACA ciphers, past and present.

So, if you’ve not heard of an Aristocrat, a Patristocrat, or a Morbit before, find one of those sources first to know exactly what kind of cipher is being discussed.

The aim of this document is to enable ACA Members to write effective computer programs to solve ciphers. For many members, writing such code is a hobby within a hobby.

Readers will find that this book makes automatic solving path much more pleasurable as many blind alleys will be avoided. Known difficulties will be noted so that they do not come as a surprise.

Except for a few sketches in Python and a little pseudo-code, there will be no actual code here. The goal is to enable coding, not to simply hand out answers.

The reader will also learn the known best practices for attacking particular ACA systems.

Because some of the methods are quite generic, they will be covered in that fashion; no point in repeating these things fifty or sixty times.

Acknowledgements

A work like this is not and should not be a single person’s effort. My thanks to BION, SCRUFFY DOG and SUSSI for their kind reviews and suggestions. The errors remain mine.

Chapter One – The Basics of Computer Solving

Since the 1970s, members of the ACA have experimented with using computers to solve hobby level ciphers. It was SCRYER, in 1995, who first indicated that most ACA ciphers could be solved.

There are two basic types of programs, Cipher Aids and Automatic Solving. Describing these generic ideas in some detail will greatly decrease the amount of description for a given ACA cipher system.

Basic Cipher Aids

The first class of program is not the focus here. In many cases, it is an elementary coding exercise to take the discussion for any given cipher system (see Practical Cryptanalysis, any volume) and turn it into a basic interactive solver (with or without a graphical user interface). The user gives the name of a file containing the cipher and then, in some fashion, inputs probable words (tips). The program places the tip and creates an incomplete trial key. This trial key is then used to show as much plaintext as possible.

The user then examines either the incomplete key or incomplete text, extends either in some fashion using the program, and the solution eventually appears. If there are relevant statistics, such as Kasiski or Chi Square, these can also be presented.

If these problems interest you or your programming skill is minimal, check out the companion volume, *Elementary Cryptographic Programming*.

Automatic Solving

The second class is more ambitious. It is possible, for the majority of ACA systems, to completely automate the solution. That is, one inputs the problem, feeds it to the program, and gets a short list of trial solutions (one of which is usually correct or so close as to be easily patched). There are several basic schemes, referenced by the individual systems, that will now be summarized here. Tips may or may not be required depending on the system.

A Preliminary – Grading the Trial Solutions

The most fundamental piece of an automated solution is to "grade" a trial answer (even if the answer is incomplete or very wrong). The *ideal* grading process produces a score that scores higher if a new trial key creates an answer closer to correct plaintext and scores lower when its answer is worse. Second best is one that mostly does that, but can also get stuck on a futile (wrong) key because the math looks good. Getting the program "off" of such intermediate keys is a major issue for some solving schemes.

Tetragram, Digram, Monogram Lists

The most popular scoring uses *tetragrams*. These can sometimes be obtained from others, but the best route is to construct your own. To do so, for *every* ACA language, especially English, collect a large set of modern text (ideally, mostly from the most recent decade, ideally, in the hundreds of megabytes range) and counts tetragram frequencies at every offset of the text. For some systems, where we assemble a pair of columns, *digram* frequencies also work well. Once in a while, *monogram* frequencies are useful (e.g. the last, standalone column in some system or other).

Where to get these "large sets of modern text" will vary over time, but the reader can simply ask around on ACA social media for directions to the best, current royalty free sources. One idea: Google on the magic word "corpus". Also, look up TIGER EYE's presentation (*Use of Free Corpora...*) in the 2020 ACA

Convention slides on the website. Then, for each language, the input is cleaned up and counted. One such *tetragram* list in English commences as follows:

TION	1130126	THES	585148
NTHE	740575	WITH	506578
THER	684987	INTH	479325
OFTH	640951	OTHE	478714
FTHE	632411	THAT	455998
ATIO	628806	DTHE	451451

The full list is quite long. Collecting one's own list (a one-time operation) is the main thing if one can't obtain a good list from a member. For technical reasons relating to probability theory, summing the log of these frequencies usually works best for actual grades. The cleanup consists mostly of getting rid of the odd Japanese or German phrase that nowadays shows up in web-based sources for "English" text. For some languages like Spanish, it means correcting the text to ACA's representation of the alphabet in English (the "tilde N" becomes N, etc.). See the *Xenocrypt Handbook* for more.

It is easy to select a *tetragram* list in a program by language – which list to use is an independent variable and means that if your solver works for English, it also works for Spanish, German, etc. because plugging in a per-language list is trivial.

Tip Bonus

If a tip is known (especially when its location is known) a bonus score can be given for trial plaintext that matches the tip in full or in part. This supplemental grade can make a large practical difference.

Grading by Plausibility

Another idea recently tried with some success is to separately grade the would-be stream for plausibility. This works best in cases (like transpositions) where the full would-be deciphered text is present. One looks to see if there are a plausible fraction of vowels (between 30 and 50 per cent perhaps) and whether there are consecutive strings of vowels or consonants (maybe five or more of either). Solutions with bad vowel consonant percentages or too many consecutive vowels or consonants are graded with "zero" or perhaps with a big reduction of the tetragram score.

Word Lists

Word lists may be useful, particularly in English. The concept is to scan the same sources as above and collect the unique words within them, ideally frequency counted by word as well. Sometimes, word lists directly figure into the grading process as a given automated scheme may produce mostly or entirely plaintext words. Useful patterns (such as ABCCBEEBDDB and cvssvssvssv) may be attached to an individual word like Mississippi, along with its frequency. (If unfamiliar with these pattern schemes, look into Pattern Word and Non-Pattern Word books).

Methods of Automatic Solution

Key Exhaustion.

This is literally trying all of the keys (that is, brute force). Suppose one already possesses an ordinary program that, given the ciphertext and correct key, produces the answer? That is a great fraction of what is needed here. To make it an auto solver, one needs an outer shell that invokes that deciphering core many times with *all* the allowed keys. One then grades each decipherment and keeps the top scoring answers in an ordered list. One of the top solutions will be the answer. Can this really work?

Yes. Some historical systems simply don't have enough cipher keys to defeat even a 1970s era computer. Today's computers can solve ACA systems with 9 factorial to as many as 12 factorial keys. It is very helpful to have a subroutine or (better) an object that creates the factorial keys directly. Why? So that key construction only costs 9 factorial rather than 9 to the tenth power. It's the difference between 363,000 iterations and one billion iterations. See FactorialKeySet.py in Appendix A for an example of how to do this. This keeps execution within range of an exhaustive solution for many ciphers.

Note also if you do FactorialKeySet, the cost of the solution is some constant times the largest period tried. Why? Because the cost of a period of 9 (9! trial solutions) is much larger than 8 (8! trial solutions).

Here's the outline of a generic key exhaustion method using the FactorialKeySet:

```
 factS = FactorialKeySet(5)  # 5! keys to process in this example.
 key = factS.start()  # required. Makes first key, copies to key.
 while (factS.hasMore()) : # Have all keys been tried?
    pt = trialDecipher(key)
    score = grade(pt)  # how much like English does the PT look?
    if(score>(bestScore*0.9)) :  # Consider "pretty good" high scoring answers.
        saveCurrentKeyAndAnswer(key,pt,score) # This is a list ordered by score
        if(score>bestScore) : bestScore=score # upgrade if really highest score
    key = factS.next()   # creates "next" trial key & copies it to key
printTopScoringAnswers()
```

An actual example has more code (particularly, the "trialDecipher" part). However, "trialDecipher" is just ordinary decipherment as the legitimate receiver of the message would do it. It's just that we don't know which key is correct.

The grade is usually done with tetragram scoring. One swaps in the tetragram object for the current language (usually English) and add them up like this:

```
def grade(pt) :
    global tetragrams
    score = 0
    for i in range(0,len(pt)-3) : # increment i by 1
        score = score + tetragrams[pt[i]][pt[i+1]][pt[i+2]][pt[i+3]]
    return score
```

One might have to convert pt[i] from an alphabetic letter ('A' or 'M' or 'Z') to an integer (0, 12, 25). It works best for the "tetragrams" table to contain the natural log of the raw frequency from the underlying data source.

If you look at a table of factorials between 3 and 15, and attach a little experience, you'll find that most systems vulnerable to this will "time out" between 9 and 12 factorial. Thirteen factorial is, obviously, thirteen times whatever twelve factorial costs. In fact, it is nearly thirteen times all the previous periods combined. There's ordinarily no need to do that much work at ACA lengths. If one does, it is time to go to hill climbing.

Hill Climbing.

This is by far the most common and useful method. An astonishingly large fraction of ACA systems solve when run against hill climbing. For some systems, hill climbing might be employed in lieu of exhaustion (Myszkowski might work this way). Others just need it. Some solve without tips, some require tips.

Here's a nearly executable, generic version of it:

```
def hillClimb(swapCnt) :
    global bestKey
    deepCopy(tgt=currentKey,src=bestKey) # tgt's cells get src's cells
    alterCurrentKeyNTimes(someTunedConstant)  # new random start key
    deepCopy(tgt=bestKey,src=currentKey)
    bestScore = 0
    for i in range(0,swpCnt):
        alterCurrentKey()
        pt = decipherWithCurrentKey()
        score = grade(pt,tetragrams)
        score = score + gradeForTip(pt,tipOffset,tip)
        if(score>bestScore)  :
            bestScore= score # leave key as-is
            deepCopy(tgt=bestKey,src=currentKey)
        else :
             undoAlterCurrentKey() # restore key to what it was
    return bestScore  # bestKey is available also
```

Figure 1. Basic Hill Climbing algorithm

This is an artificial intelligence idea. The basic strategy: 1) Create a new trial key pseudo-randomly and also remember it as the best key. 2) Fall into a loop that runs *swapCnt* times. 3) Pseudo-randomly adjust the prior trial key (usually by pseudo-randomly swapping a pair of key letters), creating a new trial key. 4) Decipher the crypt with the new trial key (*decipherWithCurrentKey()*); grade the resulting plaintext (*score*). 5) if the new trial key grades better than the old one, keep it as the new best key (*deepCopy*) and resume at step 2. If not better, undo to the prior key and resume at 2. Thus, the key starts with some new, random value and, in the best case, is slowly altered to the actual key.

Some hill climbs fail to do this in one "go". This leads to Figure 2.

```
ans = {} # Python key/value pair to record solutions
instance = 1
swapCnt = TUNED_CONSTANT   # trial and error sets this per system.
for qq in range(0,TUNED_HILL_CLIMB_COUNT) : # trial and error
    score = hillClimb(swapCnt)       # hillClimb is the key subroutine
    pt = decipherWithBestKey()
    if(score>(bestScore*0.9)) : # accept "nearly good" answers
        uniqScore = (score*10000)+ instance  # 10000 is semi-arbitrary
        instance=instance+1 # allows multiple answers with same score
        ans[uniqScore] = (pt+" "+bestKey)
        if(score>bestScore) : bestScore = score
printBestFiftyScores(ans)
```

Figure 2. "Shotgun" or Multiple Hill Climbs.

Repeating the hill climbing operation "often enough" overcomes the problems with single hill climbs. Some call this "multiple" operations method "Shotgun". "Shotgun" is done by collecting the hill climb results, sorted by highest scores.

The first description of these ideas in the ACA was by SCRYER (who is the parent of ACA automated solving) in the ND95 issue, p. 12.

Because this is a statistical process, debugging can be very difficult. One has several things to adjust, including the number of swaps per hill climb attempt and the number of total hill climb attempts. The answer, even to a known text, can simply fail to appear at first. And then, suddenly, with just the right constants, it works splendidly ever after.

Something that might help debugging is to make sure the pseudo-random generator is high quality. Not all computer languages offer good ones by default (C does not). Using the lower order bits of some generators (eg. *randomNextInteger() MOD 26* where MOD is the remainder from schoolroom division) is not always fully random either. See Appendix B for a further discussion of this unexpectedly interesting topic.

Another idea is to have a test text larger than ACA lengths (perhaps a thousand characters). It will be easier to get working constants for the hill climb, but they still will need adjustments in most cases to work at ACA lengths. However, if long texts solve, the reader knows the basic coding was correct and can proceed to make smaller test cases, knowing that it should only be a matter of adjusting some internal constants, not raw logic.

Simulated Annealing

This is an additional idea, incorporated into a hill climbing process. Simulated annealing is adapting an industrial technique for cryptographic use. In straight up hill climbing, the pseudo-random process may get "stuck" on a fairly good grade. Why? Because follow-on key swaps do not improve the grade, so the key never changes and thus never advances to solution. In some systems, a little backtracking to a not-as-good solution will get the code on the right path. As above, try Multiple Hill Climbs as the first line of attack. But if that fails? Code up a simulated annealing function. To add it to figure one, simple change *if(score>bestScore)* to *if(score>bestScore or annealingFunctionIsTrue())*.

For especially difficult systems (Playfair and Phillips to name two), essentially *all* hill climbs get stuck on a "fairly good grade". Simulated annealing defeats that using some criteria to occasionally accept new keys with reduced grades. If not done too often, and done more earlier than later, this can get the trial solution "off" of the "fairly good grade" and then sometimes get it to the correct path. The adjustment may need to happen a couple of times. Debugging is even more difficult than straight up hill climbing and published examples may have hidden dependencies on particular tetragram lists and their magnitudes. As before, it fails and fails and then suddenly works. A few systems resist even this advanced technique. Not all ACA systems auto solve; at least not yet.

For more on Simulated Annealing, see ANCHISES, SO2007 p. 12, MJ2007 p. 12 and also the variant idea ANCHISES calls CHURN in the MJ 2009 issue.

Word List, Ciphertext

To try this method, the cipher system must preserve word divisions. One then reads in the cipher, sorts the cipher words in some fashion to order "vulnerable" words from most to least vulnerable, and then tries each word recursively using the word list collected under "Grading" above. So, each qualified word in the list is tried for the most vulnerable cipher word and then for every one of these candidates, try out the second most vulnerable cipher word with all of *its* words. Then the third word might be tried. This can work exceedingly well when the number of qualified words in each word list is small (maybe 100 or so). After the first one or two words, it is often the case that the remaining words have partial solutions, which eliminates most in the later candidate words for those later words (whose lists are often larger). At some point, a practical limit is introduced (millions and even billions of trial solutions may routinely be possible) and the solution, as far as it exists, is graded with the partial answer produced in an ordered list.

While the solution shown is not always complete, good plaintext word candidates usually accrue. The code should allow introducing these as secondary tips at the head of the "vulnerability" list, which can cut down drastically on recursion counts and reveal more text in follow-on invocations.

Word List, Keys

Here, the idea is that we assume or hope that the problem constructor used only single words to construct whatever the key was (maybe a Polybius square, maybe two Polybius squares as in Foursquare, maybe a Fractionated Morse key). We then create the trial keys (maybe using route transpositions to cover all the bases in Polybius systems) and try them out, grading the resulting text. When two Polybius squares are involved, the technique usually alternates between them, arbitrarily trying all keywords against the first square, keeping the best result, then trying all keywords against the second square and keeping that result. When it works, it can solve systems that otherwise resist computerization. Of course, a key like ILIKEAPPLES will almost always defeat this scheme.

Some Alphabet Definitions

The keyed word list attack brings up a need for a few terms. Throughout there will be two concepts that need a name. The first will be called a *standard alphabet*.

A standard alphabet will be an alphabet, usually reordered by a keyword:

```
REPUBLICADFGHJKMNOQSTVWXYZ
```

The alphabet might be shifted (`XYZREPUBLICADFGHJKMNOQSTVW`). If it is a type the ACA calls "mixed" (see K2M in the ACA and You), it might not look like much of a keyword at all.

The second will be called the *trivial alphabet*, which is just a plain old alphabet:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

For many systems, the *standard alphabet* works "against" the *trivial alphabet* in a manner the system describes/requires. The fundamental issue is recreating the *standard alphabet*, in whatever form it takes, *vis a vis* the *trivial alphabet*.

There are Polybius equivalents of these, written in any route by combining I and J on the examples above. The trivial Polybius square is usually written in horizontal order, left to right using the trivial alphabet.

On Computers and Computer Languages

The short answer is: Use whatever you like.

As this is written:

1. There is a nice little computer called the Raspberry Pi 4B that fits in one's hand and is more famous for using low power than high performance.
2. Python is a popular computer language but, like the BASIC of old, it is interpreted and thus a bit slow.

Yet, with two to four months of available solving time per issue, even this pairing, which might be the slowest combination anyone could currently choose, would be "fast enough" to solve the ciphers of a given issue, several times over.

So, the most important criteria are for a given member to use those language(s) with which they are most familiar. Even a ten-year-old Intel PC is plenty fast enough.

We could use multi-threading techniques in our programming, but almost none of us wish to. It isn't needed. See also "Multi-threading" and "Performance" in Appendix B.

So, really, use whatever you like and there should be "enough" horsepower to get the job done.

How Particular Cipher Discussions are Organized: Main Technique, Implementation Issues, Failure Rate

From here, we discuss particular ciphers in the chapters to follow. Each cipher will have a short description of the main solving technique (see above), particular implementation issues and the relative failure rate. If "failure rate" does not appear, the cipher is an overwhelming favorite to either solve automatically with no mistakes or have a very close to correct solution that is easily fixed.

It will look like this:

Cipher Name

Main Technique: Hill Climbing

Implementation Issues: Works only in English at present

Failure Rate: Works about half the time.

Here will follow a brief or maybe lengthy description that outlines how to code up your own solver.

Chapter II – Simple Substitution

This is the most basic, but perhaps the most important auto solver. Once you write and debug a simple substitution solver, these things happen relative to your solving ciphers in The Cryptogram:

1. You can solve all or nearly all of the Aristocrats and Patristocrats. About eight Xenocrypts, too.
2. You understand the basic framework of how to solve the rest.
3. Some ciphers, such as Checkerboard and Monome-Dinome, can be converted to Simple Substitution and solved as Simple Substitution.
4. The total solution count is approximately 45 solutions per issue. For one program.

Though it is not immediately apparent, basic Simple Substitution is also the most forgiving from a debugging point of view. This program has a very high solution to effort ratio. It really should be the first method attempted all around.

That said, for the really exceptional problems, more than the base program might be written. The basic program will solve all but a handful (the A-25 or P-Sp-2 types mostly) and allow plenty of time for a by-hand solution for those few. Or, one could write a specialty program or two just for those. In the latter case, this becomes one of a series of similar programs in the end.

Simple Substitution With and Without Word Divisions (Aristocrats and Patristocrats)

Main Technique: Hill Climbing

Implementation Difficulties: Dealing with badly distorted texts.

Failure Rate: Eventually, only a small handful do not solve.

The Introduction provides most of the discussion necessary for implementation.

The first and most important thing to realize that it is not necessary to distinguish between Aristocrats (ciphers that keep the spaces between words) and Patristocrats (which do not) with an ordinary hill climber. The Aristocrats can have their word spaces deliberately removed (as should obviously be done for Patristocrats). Both get fed to the same solver.

It is helpful to implement the score bonus for correct tips. Even though Aristocrats don't have tips, a failed run may still reveal a word or two. Using this as a tip on a second run often yields the solution.

Note that a tip bonus may take two forms: Where the tip is precisely place and where it exists "somewhere" in the text. Both can be managed with the same routine – just use -1 as a trigger for "we don't know exactly where the tip is" and grade the best tip location found. It costs more, but it still is a big plus – too large an advantage to surrender.

Another thing is to realize that your program need not care about the various alphabet types (K1, K2, K3, K4, K3M as seen in *The ACA & You*). Just hold (say) the plaintext alphabet to ABC…XYZ and vary the ciphertext alphabet. One will get an equivalent key which can be readily converted to (at least) K1 and K2 keys if one wishes. If there is no keyword at all, it doesn't affect the solution.

Note that successful solutions may not have perfect keys. Suppose the solver gives a letter perfect solution, but the actual plaintext has no B or Q in it. In that case, there are two equivalent cipher keys; one that gets the cipher for B and Q right and another that gets them wrong. The answer is still letter-perfect. Similarly, a solution with one or two letters wrong can usually be understood; but there are multiple possible keys, nearly the same, that produce that result.

Checkerboard and Monome-Dinome

It is trivial to recast a checkerboard as a Patristocrat

For Monome-Dinome, one must identify the “row digits” for the two digit substitutes. Once done, it is very easy to translate the ciphertext into a Patristocrat with arbitrary cipher letters assigned to each letter, because we know which letters are represented by one digit and which by two.

To find the row digits, take a frequency count of numbers. The highest value is virtually always one of the row numbers. So, the code should assume that is a row number. The second can be determined as follows:

- If a digit is ever doubled, it cannot be a row indicator.
- The final digit cannot be a row indicator.
- Any digit that is a right contact of the known row digit cannot be a row digit.
- Any digit that is a left contact of the known row digit cannot be a row digit.

The highest frequency remaining number is the second row digit. This has been successful every time tried so far.

6x6 Patristocrats

For some 6x6 ciphers, such as a 6x6 Checkerboard, a special form that uses a 36 “letter” alphabet may be employed. It is not necessary to try and force the numbers where they belong. The text may thereby be a bit off (especially, numbers may be wrong), but this is easy to adjust in practice.

Patristocrats With One “Known” Letter

Because of the large number of problems in the Cryptogram, the problems nearer the end (A22-A25, P-Sp-1, P-Sp-2) often contain texts that are deliberately distorted, albeit still in English.

The beauty of tetragram scoring is that even distorted texts often solve. But for those few that don’t, this trick often works: Run the solver 26 times, but force the most popular letter to be each of the 26 letters, one per run. This adjustment can be enough to nudge one of the runs to solution. Sure, it runs 26 times longer, but when it works, you have an answer, probably in less time than setting it up for by-hand techniques like a Consonant Line.

Alternate Aristocrat Solver

Main Implementation: Wordlist, Cipher

Implementation Difficulty: Excessive recursion leading to incomplete solutions

This is an alternative solver good for Aristocrats only. One collects a large-ish word list in the language of choice. Then, the cipher words are graded for vulnerability. Because it is simple substitution, the longest words with the most repeated letters are the most vulnerable. Words with multiple repeats segregate into word lists with fewer words than those of the same length with few repeats. In fact, the ACA Pattern Word concept can be used to make particular lists smaller still. (ABCC is a different set of words than ABCB).

One simply proceeds in "vulnerability" order, recursively trying each word for the most vulnerable word, the recursively go on to the list for the second-most, and so on down the line. Starting with the second word, the prior assignments shrink the possible keys and also further constrain (and shrink) the word list. One can also order the second and subsequent words not just by lengths and repeats, but also by the number of letters in common with earlier words on the list.

When it works well, the number of candidate words for "the next most vulnerable" word shrinks to a tractable size in all cases and eventually the solution appears.

In many cases, though, the number of combinations can explode (if the word lists don't shrink much). This requires a practical implementation to select an overall recursion limit. Once reached, a partial solution is evaluated and put into an ordered list.

The partial solutions may reveal actual words. These can be forced to the head of the line and be treated as the one and only word on the list for that particular word. That constrains the rest of the words in the list enough that the combinations are now reasonable in number and lead to the full solution after a pass or two.

Chapter III – The Null Cipher, The Baconian, The Keyphrase, The Tridigital

These systems do not fit in well with other systems and so will be covered here. On a paper-and-pencil basis, they are among the easier systems; they actually are among the more difficult to auto solve.

Null Cipher

Main Technique: Exhaustion

Implementation issues: Irregular, novel schemes.

Failure Rate: High if the technique used for encipherment is not anticipated.

The advice in Practical Cryptanalysis (originally Cryptographic ABCs) is to the point: Be willing to try anything. Null ciphers have a great many possible construction techniques and novel ones appear with some regularity.

Popular methods include:

1. Some sort of trigger letter or letters to establish that "the next letter" is part of the plaintext. These are fairly rare, because constructing them is fairly difficult.
2. Some sort of "game" with the individual words. Easy methods include taking the first or last letter of the word. More often, it is a pattern; maybe the 3rd letter, the 2nd, the 1st, and then the 2nd letter. This pattern is then repeated. These kinds of ideas can be used from left to right or from the end of the word right to left. It can even be a combination.
3. Some sort of game that ignores word divisions altogether. Here, it might be every fifth letter. It might again be some sort of short pattern; maybe the 5th, the 4th, the 1st, the 3rd, the 3rd again and then repeat.

The problem with computerizing Null is twofold.

1. You have to anticipate most or all of the various methods that might be actually used and write code to try them out. If you don't anticipate the method, you tend not to get a solution at all.
2. Your grading scheme must account, somehow, for the variations in text length. It is *not* usual for trial ciphertext to vary greatly in length. In Null, it is a critical problem. The right answer might be buried in dozens or even hundreds of well-scoring trial solutions, especially those with more "plaintext" due to the assumed scheme.

Recent efforts at Null have suggested plausibility checks (ensure the text is 30 to 50 percent vowels, does not have strings of five or more consecutive consonants, five or more consecutive vowels) greatly reduces the "noise" and makes the solution more accessible.

Baconian Cipher

Main Technique: Hill Climbing

Implementation Issues: Shuttling letters between two classes.

Failure Rate: High if irregular enciphering method chosen.

The Baconian has many of the same issues as Null, but it is leavened by the fact that it is a super-encipherment of a simple substitution where each letter is replaced with a five bit, digital number (the classic “a” and “b” can be set to “0” and “1” very usefully in code; it also clarifies what’s going on).

It then simply becomes an exercise of deciding how the 0s (a’s) and 1s (b’s) were divided up.

One scheme is to have N hill climbs and each particular climb has a fixed number of letters assigned to 0 and 1 in each bucket. The hill climb swap pseudo randomly swaps a pair of letters between the buckets.

The generality of this approach defeats many custom schemes. Maybe the constructor decides to divide based on whether letters have curves in them or all straight lines. This scheme will defeat that concept without even knowing it is happening.

Practical Cryptanalysis talks about using only some of the letters of the words, but this approach is seldom seen as it takes a great deal of magazine space. Generally, all letters participate in the solution, one bit each, but that idea it is something to watch for.

A helpful idea is to see if the text, however you process it, should always produce total “bit” candidates divisible by five.

KeyPhrase and Tridigital

Main Technique: Wordlist, ciphertext

Implementation issues: “Grading” vulnerable words, excessive recursion.

Failure Rate: Fully automatic solution is not typical. Success rate good if one does multiple iterations, adding new “known” words. The total iteration count may be too high to progress in some cases.

These ciphers have a lot in common, so will be covered together. Key reconstruction is completely different – in Keyphrase, we are looking for a 26 letter English sentence or phrase. In Tridigital, we look to discern two things – which number is the “space between words” and then how to reorder the digits to produce the keyword. So far, that part is still done by hand.

In Keyphrase, word divisions are given. In Tridigital, the digit that is the word separator must be separately computed. Once correctly computed, we have the word divisions and so in either cipher, we can apply the word list to ciphertext. Usually, a tip of at least one cipher word is given and this is extremely helpful.

Tridigital Space Identification

Consider this fragment of a Tridigital published in the January-February 2021 Cryptogram as problem E-18: **83264 36842 16641**

We see the digit “6” at positions 4, 7, 12, and 13. The gaps are thus 2, 3, and 0. The minimum for this short stretch is 0 and the maximum is 3. Now, if 6 is the space character, we have a contradiction. It appears twice in a row. So, it cannot be the space. Similarly, we can count the minimum and maximum gaps. Very large gaps between appearances of a number demonstrate it is most unlikely to be the space. So, the minimum and maximum gaps, for a given number, tell if it can be the space or not. Now, let’s apply these ideas to the whole cipher, not just a convenient fragment.

The minimum and maximum for the whole cipher JF 2021 E-18 is given here:

Cipher Digit	Min	Max	Val(Min)	Val(Max)	Score
0	121	0	0	0	0
1	0	27	0	0	0
2	0	45	0	0	0
3	2	19	3	2	6
4	2	9	3	3	9
5	5	22	3	2	6
6	0	19	0	2	0
7	1	14	3	3	9
8	0	18	0	2	0
9	34	34	0	0	0

Table 1 Min and Max Gaps for Sample Cipher with Score

Let's describe the "Val" function, something we use (results seen in Figure 1) to produce the score for each digit to assess its ability to be the space character:

Min or Max	Val(x)
0	0
1-15	3
15-26	2
>26	0

Table 2 Scoring Function for Tridigital Gaps

All "Val" does is give a mathematical score to "good gap" versus "not very good gap" and "no way this gap qualifies". As you can see, in Figure 1, we captured the minimum and maximum value for each letter. Applying "Val" to each Min and Max, and multiplying them together, produces a score that is a figure-of-merit for whether that digit could be the space or not. Note that, uncommonly, cipher digit 0 does not appear in the JF2021 cipher. However, the initial values (seen in the table) lead to a zero score.

Table 1 demonstrates there are a few candidates for the space character; two with the top score of nine. It turns out that 4 is correct. The top scoring values, almost always with at least one of them being nine, is the correct space value. At worst, they can be tried in order of scoring, but inspection usually shows which high scorer is the better choice. The lower maximum between top scorers is usually a good hint as it implies a likelier distribution of word sizes.

As SUSSI reminds the author, the very first digit cannot be the space and the last is less likely to be the space (by ACA convention, by real life practicality). So, digit 7, while high scoring, should be downgraded a bit in practice. Digit 8 is doubled anyway, but it is still disqualified on these grounds.

Word Vulnerability Issues

Once Tridigital has its space correctly identified, the concerns and coding are at least generically similar. The idea behind a wordlist-based auto solver is to gather up the cipher words. They are not processed in order, but by their "vulnerability". That is, by their ability to constrain a solution.

If a single word tip is given, which is nearly universal, it is artificially sorted to become the "most vulnerable" word once correctly identified.

In Aristocrats, we gather up the word list and grade it two ways – one grade by length and the other by the number of repeated letters it has. Words with higher values of both make better, more vulnerable words to try (their list of candidates is shorter) and the list of vulnerable cipher words are ordered accordingly.

In Tridigital and Keyphrase, a lot of words survive because the multiple substitution makes it ambiguous whether cipher repeats are significant or not. If we have this cipher word: 92236 or, in Keyphrase, EIITB, we know that both APPLE and YEARS qualify. However, SHUNS does not. In practice, then, a lot of words qualify; more than in (say) an Aristocrat solver implementing the same idea. In an Aristocrat, EIITB could not be YEARS.

Still, one should order by word length, at least, and words that are impossible because of prior assignments should be rejected.

That is, the word lists should be gathered and the *recursively* applied (see Appendix B). An informal cut-off should be implemented as the lists could be large and the computation cost is the size of the "most vulnerable" list times the size of the second most vulnerable list and so on. At some point, you cheat a bit, abort, and print the answer(s) you have based on best tetragram scores.

The game then becomes spotting new plaintext words and feeding them to a second invocation. As all tip words or solved words are "lists" with a length of one and because they go to head of the line, more of the plaintext is seen and more is constrained. Eventually the solution appears.

Chapter IV – The Rail Fence, Redefence, and Route Transposition Ciphers

This chapter covers several popular, elementary transpositions.

Rail Fence and Redefence Transpositions

Main Technique: Exhaustion

Implementation Issues: Pre-filled Rectangle.

Railfence was adopted primarily because of its real world use during the US Civil War. The key space is quite small and so provides a very limited set of solutions for a computer to exhaustively examine. The slightly upgraded Redefence contains half of the Rail Fence key space. One could write a single program to solve both ciphers at once and probably not notice the performance difference or the effectiveness difference.

The cipher, which is intended to represent "rails" in a 19th century fence, has a "W" or an "M" appearance, depending on where things commence. Here is a "period" of three:

```
   *        _
 *   *   *    _
*      *       _
```

This is the "complete" "M" form as far as it needs to go. One would replace this outline with plaintext (if the text was only nine characters long). The asterisks show the variations in starting location and the minimum unit of replication. Note the the "W" form is simply the "M" form with the first two letters omitted (in the case of period 3).

The encipherment can commence at any of the asterisk points. The cipher can also be written in left to right or row by row by the constructor.

Redefence takes this a little farther by taking off the rows in a keyed order. So, there are six additional decipherments for Redefence in period 3 (three factorial).

Solving is relatively straightforward – simply try all combinations and grade the result. Note, too, that it is well and sufficient to create a standard two dimensional array of characters and make it as wide as the plaintext. One then fills in the "M" or "W" elements (which are few enough to be precomputed, then replicated to fill the rectangle). Once this is done, it is a cinch to write code to place the ciphertext into the rectangles (you merely ignore the spaces) and then take off the result as plaintext as the letters have replaced the asterisk or whatever non-space one employs.

This method, with extensive numbers of spaces, may sound a little inefficient, but on a modern computer, it is lightning fast and easy to debug. No more fuss is needed for this one.

The Route Transposition Cipher

Main Technique: Exhaustion

Implementation Issues: Long dull slog, four corners.

The route is another classic transposition with too small a key space. Once one determines the correct rectangle size(s), one simply tries out all routes for each one. There is enough symmetry that sometimes, two different routes produce the correct answer. Depending on your implementation choices, you may only have to do one version of a rectangle pair.

Suppose you have a 75 character text? Well, you are looking at 3x25 and 5x15 rectangles. Done in a certain style, you might not have to do 25x3 or 15x5. But how? Consider that the routes may commence from any of the four corners. There is a reverse spiral, but that is implemented by doing a regular spiral but instead placing the text from last CT letter to first instead of first to last. So, in the end, if you commence in all proper directions from each corner, symmetry should give you all of the routes for a 15x5 when you use a 5x15.

The routes are:

Horizontal

Alternating Horizontal

Vertical

Alternating Vertical

Diagonal Left

Alternate Diagonal Left

Diagonal Down

Alternate Diagonal Down

Spiral Clockwise In

Spiral Clockwise Out

Spiral Counter-clockwise In

Spiral Counter-clockwise Out

“Alternate Diagonal Left” and “Diagonal Down” require a little explanation. To allow oneself to process only the 5x15 block, one must commence diagonals in particular directions. The “left” versus “down” refers to placing the second letter of the plaintext when the encipherment starts in the upper left corner; down or left from the corner. The rest of the transposition flows fairly obviously from that point. In the other corners, the directions might need adjustment, but there will be a direct analog to the upper left corner case.

Note the symmetry here; if the actual square chosen by the encipher was 10x3, then a vertical encipherment will produce the same results in that block as a horizontal one does in a 3x10. If you look carefully at a complete example, you will see such symmetry carried through. This matters because now we must also account for the four corners.

There are two obvious choices: 1) Program all forty eight versions of the routes. 2) Program the twelve routes above and then “flip” the cipher along the horizontal axis, the vertical axis or both. That will get the cipher block in the other corners. Simple example of ABCDEF enciphered as a 2x3 block, then flipped in the various ways:

No Flips	Horizontal Flip	Vertical Flip	Both
ABC DEF	DEF ABC	CBA FED	FED CBA

As can be seen, the letter A, the first letter, ends up in each of the four corners. Because of the symmetry involved in the above definitions, this means that if the twelve routes are performed correctly, then the final result is flipped, we will have programmed all forty eight combinations and accounted for the 3x2 block by symmetry.

Double route transposition is commonplace and usually unannounced. But, if the forty eight transpositions are managed, then repeating the process will get the rest. It may be necessary to implement “going in” and “going out” forms to account for the reality of how double transpositions work.

Now let’s do some counting. For a 72 letter text (see Appendix A, Factors), then there are five individual blocks with the lead digit the smaller. That is, the block might be `2x36, 3x24, 4x18 ,6x12 or 8x9`. But, by symmetry, we don’t have to worry about the ones that commence with the larger dimension. So, that’s 5 x 48 total trial encipherments, which the worst case for known lengths below 100 and it is only 5 x 48 x 48 for the double route case. On today’s computers, producing them all is eyeblink speed; sorting them in scoring order to look at only the best answers is likewise fast.