

February 16, 2003

LEGAL NOTICE

The information contained in this document is previously made public in Patent Applications PCT/SE99/01740; EP 98118910.3 (English).

LEGAL NOTICE

An implementation of the technology contained in this document may be subject to patent or a patent application. Please contact Protego Information to obtain a licence, the latest version of the software, and help with the target implementation.

The HCIA Cipher Technology

Introduction

Ciphers are constructed to prevent a cryptanalyst from obtaining the secret plaintext or a secret key by a cryptanalytic attack on available non-secret ciphertext. A cipher system is normally constructed using documented design criteria aiming for a convincing argument that no reasonable cryptanalytic attack may work. This put the cryptographer in a position where he must essentially be more skilled than the cryptanalyst.

The cryptanalyst, however, normally attack not only a specific instance of a cipher but a network of nodes connected by encrypted links protected by the target cipher. He may also make use of research, inventions, and the general advance of technology that has taken place between the point in time, when the cryptographer designed the cipher, and when the cryptanalyst perform the attack.

These conditions make the work extremely hard for the cryptographer. Due to this the current state of the art is to publicly disclose a proposed cipher system and then wait, possibly for decades, until a scientific consensus can be reached that the proposed cipher system is secure.

The above model -- that the cryptographer may design and analyse a cipher so well that no cryptanalyst will be able to exploit any weakness ever -- is not the only approach to the encryption problem. In this report we describe a cipher that is based upon a different meta-model. Using HCIA neither the cryptanalyst nor the cryptographer will be able to perform detailed analysis of the cipher system. This is due to that a generator model is used where an instance of a cipher system is produced at runtime, kept secret during encryption, and then discarded immediately afterwards.

It is clearly evident that the encrypted communication link will be secure as long as the cryptanalyst is not successful. We therefore argue that the level of knowledge, that the cryptographer can obtain about the cipher during the design phase, is not really important. What is important is an assessment on the degree of knowledge that the cryptanalyst can obtain about the cipher, in future, during an attack.

The above argument may hold only if the HCIA implementation is fundamentally different compared to conventional designs. In particular the conventional designs will be called "cipher algorithms" and it will be shown that the HCIA system cannot be constructed using any such algorithm.

It is previously known, and well established, that there are some computational problems that are impossible to solve, or uncomputable. It will be shown that the problem that the cryptanalyst try to solve during cryptanalysis is uncomputable, blocking the main and most important avenue of attack. This fact force the cryptanalyst to use different and simpler attacks, that the cryptographer may easily block, up to any desired level, using a set of simple design rules.

Protego Information AB
<http://www.protego.se>
E-Mail: cryptography@protego.se

As the HCIA has leveled the field -- the cryptanalyst may no longer take advantage of future research and technology advance -- the cryptographer is in a good position to complete his work successfully. The HCIA technology essentially remove the cryptanalyst from the design parameters, and clear cryptography from a sense of mysticism. This opens the possibility to design ciphers adopted to other design parameters such as message format and length, encryption latency, throughput, adoption to project-specific hardware and software requirements, or making a cipher that perform well both on a hardware implementation on a server and on a thin client as DSP/microcontroller software.

In practice the level of security that can be obtained is limited by implementation issues. This is almost always the case even if a conventional cipher algorithm is used, as the field of implementation issues is currently in desperate need. We hope that the HCIA cipher system, that make it comparatively easy to design a mathematically secure cipher system, may increase the interest in research on the difficult implementation issue.

A Model of the Cryptanalyst

In the discussion below it is important to understand that before we discuss the HCIA system, a simplified and principal discussion must be held. We will in particular discuss if certain tasks can or can not be performed with a specified amount of resources. In particular, we warn The Reader for the following trap:

Any cipher that works must have a key, a string of bits. Independent of how the cipher is constructed, the cryptanalyst can simply take a guess of the key, and succeed with a probability greater than zero. In the discussion below, disregard this particular attack! The simplicity of the attack suggests that analysis of it may also be simple. It is not; it will be treated in detail at the appropriate place.

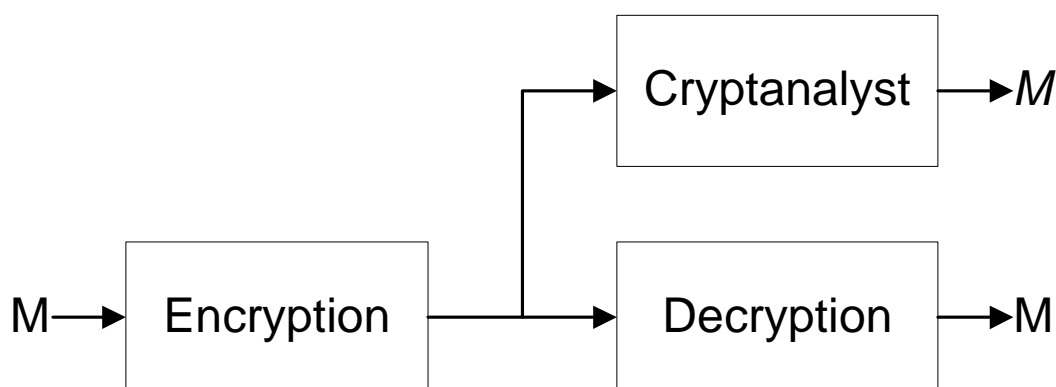


Figure 1. Encryption, Decryption, and a Cryptanalyst [137].

In figure 1 above, the cryptanalyst is introduced merely as a blackbox. In reality, we cannot possibly know much more. First, the cryptanalyst works in secret. If a break or other weakness is found, it can be exploited only if the legitimate user of the system is kept in the dark. Otherwise would the cipher be changed into a more secure one, or merely any other cipher. Second, the cryptanalyst will apply the attack only when the system is in operational use. The design of the cipher occurs much earlier.

Possibly some development in mathematics, electronics, quantum mechanics, or optics could be used as a new tool when attacking a cipher. Clearly, this is an advantage for the cryptanalyst.

Finally we note that the legitimate user tend to treat the encrypted channel as perfectly secure, and trust the channel with extremely sensitive information. Because of this the cryptanalyst may mount a much more expensive and powerful attack compared to the the level of protection the cipher was originally designed for. [156 p 25 sec. "Kryptobehov"]

Nevertheless, we proceed with an investigation of the capabilities of the cryptanalyst, and we start with a simple input-output relationship:

The cryptanalyst, when cracking the code, reads (or obtains) the ciphertext. The ciphertext, that is encrypted, cannot reasonably be kept from the cryptanalyst. The purpose of encryption is that the encrypted ciphertext can be communicated using insecure channels. Possibly the cryptanalyst will not obtain the complete ciphertext, as there could be a disturbance that prevents the cryptanalyst from reading some part of the ciphertext. In the discussion below we assume that the cryptanalyst can obtain the ciphertext completely, as he can himself discard this information should this be an advantage.

The cryptanalyst may also obtain some part of the secret plaintext. Guessing is an effective means to obtain the plaintext, as in most cases application specific files are sent, and almost all files contain segments (the "header") that is invariant. The cryptanalyst will also know the "statistics" of the plaintext consisting of the relative frequencies of the symbols of the plaintext. The used cipher key is more problematic. We cannot assume that the cryptanalyst knows the key, but in some situations, when cipher keys are generated using pseudo-random means, the cryptanalyst may also obtain partial knowledge of the cipher key.

We also assume that the cryptanalyst has obtained "side information" consisting of the context in which the messages are being sent and other information that may be helpful for the cryptanalyst.

The purpose of cryptanalysis, and so being the most preferred output, is the secret cipher key. A successful cryptanalysis could also be mounted on a model of the cipher yielding the secret plaintext without explicitly obtaining the original cipher key. If the cryptanalyst cannot obtain the cipher key exactly, he may obtain an approximation of the cipher key. If the approximation is good we say that the cryptanalysis was successful. The cryptanalyst could also obtain the key probabilistically so that the correct cipher key is obtained only with some probability. We say that the cryptanalyst has successfully broken the cipher if the probability is large compared to random guessing.

The cryptanalyst could also, depending on the attack model, obtain some other kind of information that the cryptanalyst himself would consider favourable for him. The above discussion on keys also applies to the secret plaintext; the plaintext could be obtained approximately and/or probabilistically. Finally the cryptanalyst may obtain the key or the plaintext in some other field or using some special symbol set [], where the approximation and probability discussion above assumes the ordinary binary symbol set.

It is difficult to formalise the activities of the cryptanalyst. But for any cryptanalyst, using any means, the above discussion show us that the input to a cryptanalytic attack consists of a varying number of variable length strings, containing a specification of language statistics, obtained plaintext segments, format specification for file headers, etc., and the obtained ciphertext. In the same way may we argue that the final result of the cryptanalysis consist of a varying number of variable length strings, one of them hopefully containing the cipher key, as discussed above, and possibly other side-reports including instructions for archive filing, etc.

In real life the cryptanalyst may also obtain information or instructions by other means, using human-human communication, rumours, or phone conversations and the like. This cannot alter the result above. Phone conversations is, today, digital streams in transit, even if presented in analog form at both ends. From a principal point of view we may also argue that human-human communication also could, in principle, be digitally recorded. Even if not all aspects of human-human communication can be digitally recorded, it is easy to see, that any information that can be of use for a cryptanalyst for code-breaking can, in principle, always be digitally recorded. Note that we are not arguing that this actually takes place in practice. But it is clear that any information that the cryptanalyst can possibly exploit must necessarily be possible to record in an unspecified number of digital streams of some (any) length, and the same argument applies to the output.

At this point it is important that it is observed that no restriction has been imposed on the cryptanalyst by this assumption. Any cryptanalyst, now and in future, obeys and must obey this law.

All inputs to and all outputs from a cryptanalysis session must be possible, in principle, to record on a varying number of variable length files. (Any technical limitations of any particular technology used is ignored).

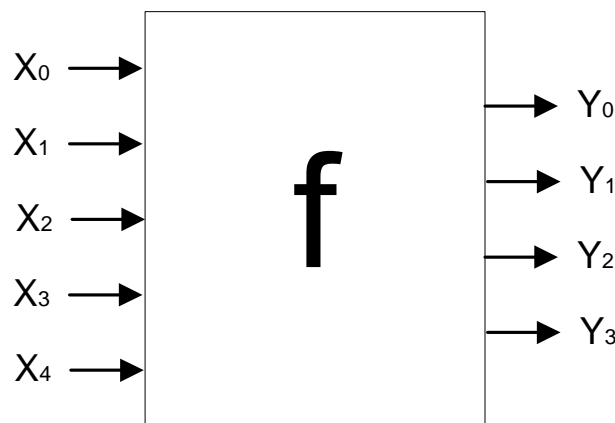


Figure 2. A function with multiple inputs and multiple outputs.

We now model the activities of the cryptanalyst by a function that maps the input information set to the output information set. The function is simply a model of the mapping, and do not suggest any measurement of the cost or effort of actually doing the real work. We note that nothing has been said about this activity. As no limitation has been posed on the function model, in the figure, it must be valid for any kind of cryptanalysis activity independent of methods used.

This model may be further simplified. We may say that the cryptanalyst, any cryptanalyst, may be modelled by an integer function. An integer function is a function that maps one integer in $N=\{0,1,2,3,.. \}$ onto another integer in N . This seems to be a simplification, but it is not. Consider the situation in Figure 2. Now, the varying number of input streams may, in principle (a "thought experiment"), be recorded on some kind of hard-disk using a file system. The number of input files may overflow the file system, and one or several files may be too large to fit on the disk. But some arrangement could be made to concatenate a chain of hard disks, or by other means, so that finally we may (again not in reality, but in principle) collect all the information on a disk or set of disks.

The input integer "x" simply consists of the binary hex-dump of the contents of the disks. The integer is clearly not some kind of intelligent enumeration of all possible efficient cryptanalysis methods. The input "x" is an equivalent information of the the varying number of variable length inputs we had above. This technique is called "Gödel Enumeration" after he who did it first. Gödel used mathematical methods when enumerating his input set. Nowadays the concept of a file system is well understood, so this model has been selected for this discussion. In passing we note that any enumeration (or model) will do.

The Reader should note that we only discuss that this could in principle be done. We can obviously never know exactly what information the cryptanalyst is actually going to use! And please forgive us for the long essay above. In cryptology it is important to look very carefully at all details.

We have now obtained the situation in the following figure:

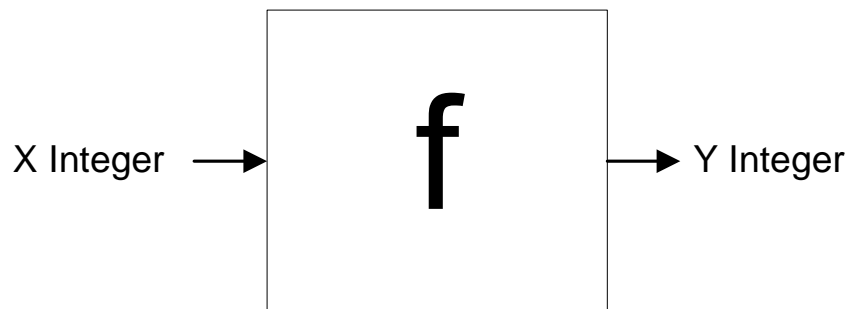


Figure 3. A function with one integer input one integer output.

(We may agree that the Reader may be curious about why so much energy has been used to arrive at such a simple formula!)

We now know that the cryptanalyst, now and in future, can be modelled without any restriction as a function that reads an integer and outputs an integer. Often, the cryptanalyst is modelled only partially. It is assumed that he applies only some kind of known functions, and the results are obtained in this restricted model. Here an unrestricted model has been chosen. The advantage of an unrestricted model is that nothing is lost -- there can never be any surprises. The drawback is that we can only say very little about the cryptanalyst. In particular we may never say anything about the contents of the function "f". But we can count how many they are!

How many integer functions are there? This is well known. But for readers that has not seen this before, we will proceed slowly in very small steps:

First, there must be an infinite number of integer functions. Is this obvious? In particular, this infiniteness is of a special kind: We say that the set of the integer functions is an uncountable set.

The set of the integer functions cannot be a finite set. To prove that the set of the integer functions is an uncountable set, and not a countable set, we assume that the set really is countable for an argument by *reductio at absurdum*.

A countable set can, in principle, be modelled by an infinite length list. So there must be a first function $f_1(x)$, a second function $f_2(x)$, a third function $f_3(x)$, and so on. By the assumption that the set is countable every possible integer function must be in the list.

Now, consider the function: $g(x)=f_x(x)+1$. If the set of all integer functions is countable, then there must exist some integer index j , so that $f_j(x)=g(x)=f_x(x)+1$. Setting $x=j$ now yields the contradiction $f_j(j)=g(j)=f_j(j)+1$.

In this context we are allowed to "add one", making the value of the function $g(j)\neq f_j(j)$. We could have added any constant. Further: this discussion is not a computability discussion. We are merely investigating the existence of certain functions. We are also allowed to use the list itself, in defining the function $g(x)$. We conclude that the set of the integer function cannot be countable. So they must be uncountable.

For the mathematical pedant we include that the above proof merely states that the set of the integer functions is an uncountable set and not a countable set. It is not strong enough to actually introduce an uncountable set.

Mathematicians may now plug their ears, because, even if proven, we include the following illustrative example: Assume again that the list exist, and that all integer functions is countable. Construct the function $g(x)=f_x(x)+c(x)$ where $c(x)\neq 0$ and where the value of the constant $c(x)$ depends on x . Using $c(1)=\{1,2,3,\dots,10\}$ we construct 10 functions $g(x)$ that is different from $f_x(x)$ when $x=1$ (by using ten different functions $c(x)$). For each of these ten functions we proceed by setting $c(2)=\{1,2,3,\dots,10\}$ when $x=2$; individually for each of the ten previous functions. Now we have a 100 functions different from $f_x(x)$ when $x=1$ and 2. We now proceed like this, and for every step in the list, that we supposed contained all integer functions, we multiply the number of functions that cannot be in the list by ten. Now, substitute the figure "ten" by any number. The countable list is of infinite length! That is how many we are talking about when we say "uncountable". It is really a lot!

We have used the powerful method of "diagonalisation" (to see the diagonal, make a function table where $f_j(x)$ are on the "y" axis and the "x" values are on the x-axis). Diagonalisation, invented by Georg Cantor in 1891 [158] [146 pp 165-166], is one of two powerful methods often used in complexity theory. The other method will be used later.

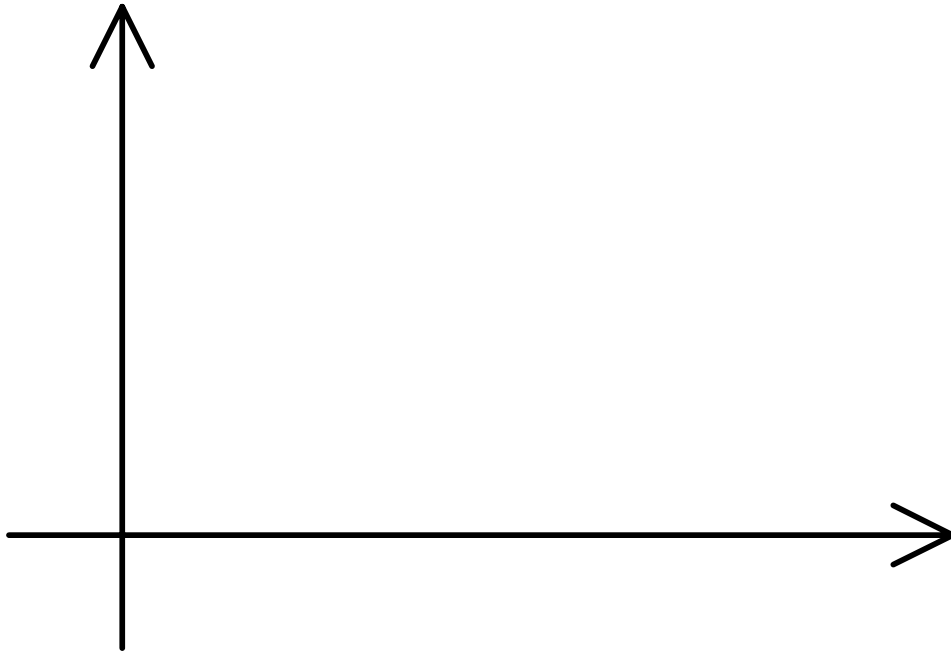


Figure 4. Picture of a plane.

A more convenient model of an uncountable set is the set of all possible points that there can be on a plane. In the picture we assume that the plane is extended infinitely in all possible directions. We also assume that the points don't cover any area; points never do. An ordinary plane (two axis and index by two independent real numbers) also has other properties, that will not be needed here.

We should now put things into perspective. We have made a model of the cryptanalyst, and we have reached a situation where the cryptanalyst may crack the cipher by applying a method, that we model with a function. We have then seen that the set of all integer functions is an uncountable set, and that an uncountable set could consist of all possible points on a surface (here of infinite size, in all possible directions, to add clarity). Now, ladies and gentlemen, that is a large number of cryptanalysis methods! Clearly the cryptanalyst cannot ever apply that many attack methods! Indeed, a much more realistic attack model would be to limit the cryptanalyst into applying only a finite number of attack functions. ("Finite" is merely less than infinity, not necessarily any small number...)

We will, however, select the following limitation: The cryptanalyst cannot possibly ever apply all functions in an uncountable set as a cryptanalysis function. The cryptanalyst must be limited to selecting cryptanalysis functions from a countable set!

Too see that the cryptanalyst cannot ever escape from the countable set, as a first step, we note that if the cryptanalyst implements the cryptanalysis using a digital computer, there must be a set of software files controlling the computer. Collecting this software and putting them on a file system yields that they can always, in principle, be converted into an integer. And the integers are countable $\{0,1,2,\dots\}$. This implies that all possible cryptanalytic methods implemented on digital computers must be countable.

The countability of processing methods is also called "computable" in complexity theory. Possible processes not in this set are called "uncomputable". Clearly, if an ordinary digital computer is used, or possibly a parallel machine with a finite number of processors, then the functional set computed (that could in principle be computed) must be a countable set.

But escaping from the countable set is no easy task! Assume that some machine could be built, possibly using optics, quantum mechanics, etc., anything!! In fact there is a branch in physics researching "quantum computations" that can compute non-trivial tasks much faster than could ever be done using ordinary (sequential) binary computers.

But any such computer must, to be of use in cryptanalysis, be built using some kind of drawing (specification). It must be operated by ordinary people (i.e. human beings). It must be possible to educate these people in using the machine for cracking ciphers. Further: the machine must be the result of a finite list of research papers (open research or secret research or any combination thereof).

The above is a list of information: instruction manuals, research papers, technical specifications. This information can be digitally recorded. Research presentations, lectures, etc., can also be digitally recorded. Finally, apply the discussion with the file system to obtain a countable set. The input to the cryptanalysis is a finite set. Hence the set of functions that can be obtained must be a countable set, no matter what. (Do we agree on this?)

That leaves us with describing the operation of the human brain. Nowadays we know much more about this, and its complexities, than we did merely a decade ago. But it falls short of escaping the countable set, that is of infinite size. That the capability of the human brain is limited by the countable set is well known as Turing's Thesis [121]. It was first published in [141] 1936. Turing's results were quickly adopted and extended by contemporary researchers in various ways [132]. The Reader is in particular encouraged to obtain a copy of Turing's paper and see another investigation of the power of the human brain (called a "computer" in Turing's paper. Back then, a "computer" was a person, i.e. a profession) (see [109]).

- -

All possible ways of attacking a cipher must correspond to the set of the integer functions, and there are an uncountable number of such functions, corresponding to every possible point on a surface. But the cryptanalyst cannot possibly actually evaluate all these functions. He is restricted to a countable subset. So, for the cryptanalyst, there must be a first cryptanalysis function, a second cryptanalysis function, a third cryptanalysis function, and so on. Let's mark these points with black spots on the plane.

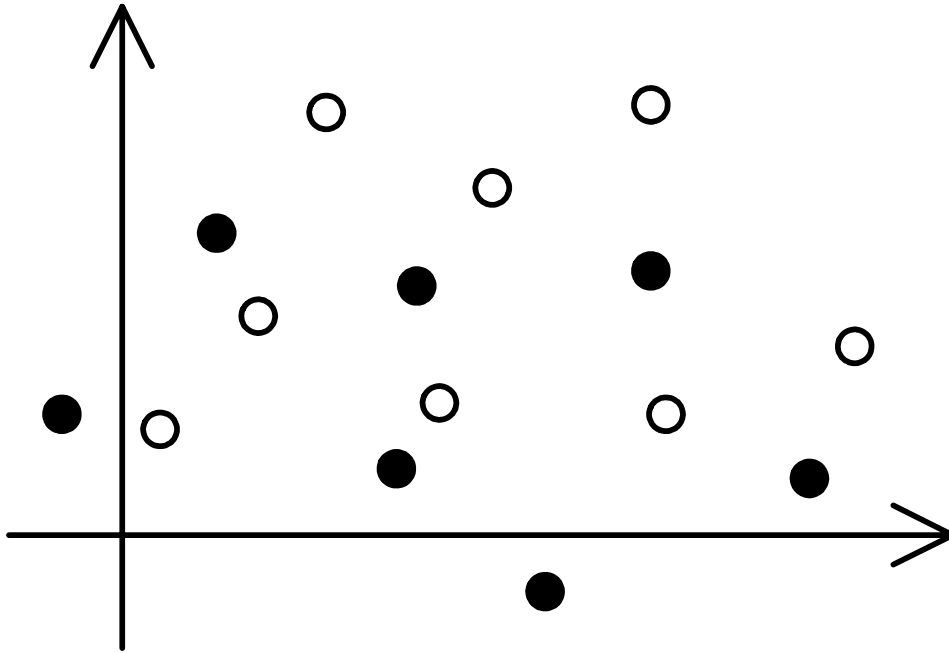


Figure 5. Picture of plane with spots. Black spots for the cryptanalyst, circles for code-breaking functions.

We now have a model of the cryptanalyst, that encompass everything that the cryptanalyst will be able to do, now and in future, independent of methods used or resources allocated. This model is a very useful tool. Here it will be used merely to some small extent. We note that even when using a very powerful model of cryptanalysis -- an infinite application of functions with computation costs ignored -- the cryptanalyst is still limited.

The model can now show us what we should accomplish with our cipher: The trick is to locate the functions (circles), that break the cipher, at some other place than on the spots where the cryptanalyst can execute codebreaking (black spots).

We see that we can have "unbreakable encryption" even if we may be certain that a cryptanalysis method exists (as a function). A function is not a computation! We also see that even if the cryptanalyst iterates forever, if no filled spot ever hit the circle, there will be no success. Ever. (It is precisely now, dear Reader, that we should temporarily forget that there must be a finite size key somewhere. Stay tune! It will come later!)

In particular we note that we don't have to be careful about where the circles will be placed. First, we have an infinite-size area to position the circles on. Second, as points don't cover any area, the probability of putting a circle exactly right on a black spot is exactly zero. This is because probabilities are evaluated using measures [159 p 1] (Kolmogorov 1933). This let us conclude that we don't really need to position the circles. Simply let "the chips fall as they may" is a way as good as any other. This is an important concept, or idéa, that becomes natural only in this setting.

While we are investigating this model, we should observe how the model works in an information-theory setting.

In "information theory" (-- when applied to cryptanalysis. Information theory is very powerful and is used also in signal theory ---) we don't investigate the difficulties involved in actually performing the

cryptanalysis. We merely look at the results. For some ciphers, called information-theory secure, not even successful cryptanalysis helps. The reason is that if the keyspace is rich enough, in comparison to the size of the set of input information available to the cryptanalyst, the cryptanalyst may end up with several "correct" cipher keys each yielding a "correct" plaintext. Based upon the information available to the cryptanalyst, he can not select the right solution. Hence the secret message remains secure even if cryptanalysis was successful!

The easiest way of obtaining an information-theory secure cipher is to use the OTP cipher (One-Time-Pad). In this cipher a key string, randomly selected using a true random number generator, is added to the secret message. In this setting any plaintext will correspond to a key corresponding to the intercepted ciphertext. The security requires that the key string is used only once, hence the name "one-time".

The OTP is an example of one of the most simple information-theory secure ciphers. In particular, using the same key twice, does not prove that the cipher is not information-theory secure (Exercise: construct a two-time pad, secure if the key is used once or twice, but not otherwise.)

In information theory the cost of obtaining the plaintext/key (one or several pairs) is ignored. This is normally interpreted as that the running time of a cryptanalysis software is ignored. In our model (Figure 5) this can only be the case if a function that breaks the cipher hits a cryptanalysis function that the cryptanalyst can reach. In general the information-theory code breaking model must be much stronger. We must assume that the cryptanalyst can evaluate every cryptanalysis function (making the entire plane black!). Hence the code breaking model used in information theory is not only "strong". It is absurd!

Nevertheless, information theory is a very important and useful tool, that will be used below. But keep the picture in mind.

Descriptions and Interpretations

The "Real World", its descriptions, and how they are interconnected have been extensively studied. Before we apply the results to our particular situation (on how to make a cipher), we shall make a few general notes.

Science, as we know of it today, consists of investigating nature. The results are then summarised in reports and on other media. This material are then stored for humanity to use in future.

In mathematics (originally a "practical" science) at the end decade of the 19th century, a branch of mathematicians started to investigate the fundamentals of mathematics itself. We have already mentioned Georg Cantor, who was investigating integers, the real numbers, and the infinity, with the goal of possibly be able to describe (prove) the continuum property of real numbers [146]. However, about 1900, due to new discoveries in set theory (i.e. logical contradictions, in particular the paradox of Bertrand Russel), it became evident that the foundation of mathematics needed a thorough work-over. Several mathematicians set out to do this work.

There where some progress as the theorem of well-ordering and an axiom system for sets (Ernst Zermelo). Bertrand Russel and Alfred North Whitehead published their "Principia Mathematica". David Hilbert, in particular, sketched a mathematical-logical program which he believed would

remove "once and forever" the problems in logical reasoning and in the foundation of mathematics known at the time. Four main questions (of 23) where (David Hilbert, Sept. 1917)[149 p 151]:

- The problem of the solvability in principle of every mathematical question.
- The problem of finding a standard of simplicity for mathematical proof.
- The problem of the relation of content and formalism in mathematics.
- The problem of the decidability of mathematical question by a finite procedure.

However, as time went by, the number of problematic questions and issues where steadily increasing. But there where also progress made: First-order logic enabled a theoretical treatment of general mathematical and scientific questions. Finally, in 1930, David Hilbert was convinced that, soon, the problems of the foundations of mathematics would eventually be solved.

Unfortunately, he was quickly disappointed, as Kurt Gödel published his now very well known result that any sufficiently rich mathematical theory must be incomplete, in the sense that there will exist mathematical propositions that may be neither true nor false [113]. In particular, the concept of "truth" could not be defined in a mathematical formal language. This indeed was serious, for mathematics. Without "truth" there cannot be any "false", hence the mathematical concept of a "proof" was threatened; a theorem can be proven when it is "true".

This is not so mystical as it first seems. An example of a mathematical entity that cannot be defined in a mathematical formal language is defining which way a rotation around a circle is in the "positive" direction. The "positive" direction is anti-clockwise. But (most) clocks go around in a certain direction merely by convention, so by replacing one convention by another, we cannot gain any advantage in a formal definition. One could even try to make a connection with the rotation direction of the Earth, but this approach would only add an additional layer of complication (i.e. which pole is the "North"?).

As given in a hint above, "infiniteness" is another entity that can not be precisely (completely) defined using mathematical theory or tools. We can always define finite sets and subsets. That is easy. We can then easily create progressively larger finite sets. But to "infinite" we can never reach using finite means. To define an "infinite" set requires another infinite set to start from, or an induction principle. Using an existing infinite set as a vehicle is obviously a circle argument. The (weak) induction principle uses an ordered list: if we have 1,2,3,4,...,n, then after n will follow n+1. The induction principle now say that this will go on forever and that there will be an infinite number of numbers. But this works only if an induction principle has been previously defined. And it is not possible to define an induction principle without referring to another induction principle. Evidently, we are locked in a circle argument. There is no infiniteness, unless you yourself would find the concept useful. Due to this reason the proof above, about the number of integer functions, can merely classify the size of the set. Infiniteness in itself cannot be circle-free introduced (created) this way.

- -

To separate the entity of the reality itself and the the descriptions that we may use to describe various properties of the reality, we use the concept of the linguistic complementarity:

The Linguistic Complementarity.

Descriptions and interpretations of a language are complementary. That is, as long as we stay within a language L, we cannot completely describe L only in it terms of its sentences -- both descriptions and interpretation processes (both sentences and interpretation processes; both models and description processes) are needed for a full account, and use, of L. However, there may be a metalanguage with higher describability than that of L, that allows a complete description of L. In that case, we say that the complementarity is transcendable. If no such metalanguage can exist, we say that the complementarity is non-transcendable. [124 sec 3 p 329][150 p iv-v]

If we as an example take a candle, it may be described in great detail. Photos may be taken, in different parts of the spectrum; the 3-D temperature distribution of the flame may be measured by optical methods, where vibration and rotation temperatures will be different due to that the system is not in equilibrium; lyric poets may write poems about the beauty of the flame; we may simply use the candle for illumination, without giving much thought on the light distribution in space or as a function of the distance to the candle.

But all the above descriptions will be partial. Even if we would add up all the reports, we can never purely from the pack of paper actually create the reality completely, i.e. a working candle, unless we actually have a candle to lit (don't incinerate the scientific reports!). Despite that the description must be partial, they can still be of great value, if we are merely interested in some specific aspect of the object.

This translates into several important situations in ordinary life. We understand that there cannot ever be a complete (perfect) car-repair manual! That must always be some aspect of the car (or the tools) unwritten in the manual, that a repair man must obtain by the use of physical objects, by experience, or by other means. Also, science will absolutely never be finished, the world will always have something new for us!

- -

Alfred Tarski: The question about "truth":

Alfred Tarski showed [140] that even if it was impossible to define "truth" completely in a formal language, it was possible to create a "higher" language. In this "higher" language the "truth" will be partially defined. But for the lower language, the "truth" definition will be complete, and the fundamental problems disappear.

A well-known example, where it is exploited that the truth is transcendable, is the Liers Paradox by Kurt Gödel:

Example (Kurt Gödel [114 Sec 7]; in [109 p 63]):

..."Suppose that on 4 May 1934, **A** makes the single statement, "Every statement which **A** makes on 4 May 1934 is false". This statement clearly cannot be true.

Also it cannot be false, since the only way for it to be false is for **A** to have made a true statement in the time specified and in that time he made only the single statement."

[114 sec 7]: ... "For consider the above statement made by **A**. **A** must specify a language *B* and say that every statement that he made in the given time was a false statement in *B*. But "false statement in *B*" cannot be expressed in *B*, and so his statement was in some other language, and the paradox disappears."

(Kurt Gödel: Note added 28 August 1963; see also [109 p 71]:

"...it can be proved rigorously that in EVERY consistent formal system that contains a certain amount of finitary number theory there exist undecidable arithmetic propositions and that, moreover, the consistency of any such system cannot be proved in the system."

Going back to cryptography and encryption we here have a problem area that would be advantageous for the cryptographer to build a cipher upon. The task of making a cipher is to find a sufficiently difficult problem for the cryptanalyst to solve. Above we see that finding a complete description to an interpretation is impossible. This problem is a well-studied problem and is well understood, and it is impossible to solve. The only thing remaining is a digital implementation.

Above we have indicated a meta-language definition for the direction of positive rotation around a circle and for the infinity. This also help in understanding the cryptanalyst's work: The problem specification for the cryptanalyst may be impossible to solve in the language specified. Then there might exist, or not exist, a higher language where these questions may be solved. This we know, and we may "arrange" things so that the "higher" language will be problematic to work in for the cryptanalyst.

- -

As for the continuum proof, that Georg Cantor was trying to obtain, that issue was indeed extremely complex, and he failed to give the answer during his lifetime:

"Though he know that his life's work was incomplete, he could not have known that, despite his own inability to come to satisfactory terms with the continuum hypothesis, no one else would succeed either. The solution remained an enigma." [146 pp 270]

"Finally, in 1963, Paul Cohen established what Gödel's Incompleteness Theorem had indeed suggested was possible. Cohen showed that neither the continuum hypothesis nor the Axiom of Choice could be proven from the axiomatic system of Zermelo-Fraenkel set theory"
[146 pp 268-269] [Cohen 1963: 147 and 148]

- -

Different Kinds of Algorithms

As stated in the introduction, we claim that the HCIA cannot be, and cannot work, if it was merely like any other cipher algorithm. In this section the concept of "an algorithm" is investigated.

The most basic fact about cipher algorithms is that one must always assume that the cryptanalyst knows the system being used [160 ch 11.2]. In any case, the security of the cipher, cannot depend on that a part of the cipher algorithm is being kept secret. The cryptanalyst may capture or buy a cipher box, and then reverse engineer the cipher. Historically, we know that the ciphertext obtained from a long stereotype message was sufficient for the recovery of the implemented cipher algorithm [105]. The only secret part of the cipher algorithm must be the secret cipher key, that can be quickly replaced.

Auguste Kerckhoffs (1835-1903, Flemish professor):

Il faut qu'il puisse [le système] sans inconvénient tomber entre les mains de l'ennemi.

No inconvenience should occur if the system falls into the hands of the enemy.

[161][160 p 196]

If the security of the system don't depend on that the algorithm is kept secret, then the designer have a choice: He may choose to publish the implemented cipher algorithm, or he may choose to keep the algorithm secret. In the research community only the approach of publishing the cipher algorithm is accepted. The Snake-Oil FAQ [155] is very clear on this point: cipher systems with unpublished cipher algorithms should be avoided, as they may be weak. A recent example of the danger of restricting access to the cipher algorithm is the debacle with the 802.11 wireless LAN encryption protocol [152]. The cipher system implemented for the 802.11 was developed in a committee, and it costed a fee to obtain the specification. Due to this no-one obtained the specification and checked out the cipher system. Then finally a competent researcher obtained the specification. Before the sun set that day, the implemented cipher had failed cryptographically in about every way possible! [153][154]

Yes, Dear Reader, we can safely agree upon that the cipher algorithm should be published. The only remaining problem is that even this issue is not at all as simple as it first seems.

First, we note that books on encryption only deal with encryption-algorithms. The encryption-algorithm is thought to be a general concept. An example, by a well-known and highly respected researcher, that has designed more cipher algorithms than most:

Prof. Dr. Kjell-Ove Widman [143 ch 3.1 p 23]:

"An algorithm is a set of formulas (and rules) describing the way a computation or a task is to be performed. A cryptographic algorithm is an algorithm prescribing the way plain text is transformed into cipher."

Even books on algorithm theory are vague on what an algorithm really is

"Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest:

"Informally, an ALGORITHM is any well-defined computational procedure that takes some value, or set of values, as INPUT and produces some value, or set of values, as OUTPUT." [107 ch. 1.1]

and

"A good algorithm is like a sharp knife--it does exactly what it is supposed to do with a minimum of applied effort. Using the wrong algorithm to solve a problem is like trying to cut a steak with a screwdriver: you may eventually get a digestible result, but you will expend considerably more effort than necessary, and the result is unlikely to be aesthetically pleasing." [107 ch 1.4, p 16]

You may compare to the algorithm definition by Donald E. Knuth in "The Art of Computer Programming Volume 1/Fundamental Algorithms".

According to Donald E. Knuth, it is required [122 Ch 1.1 p 4-6] that an algorithm meets the following five criteria:

- 1) Finiteness. An algorithm must always terminate after a finite number of steps.
- 2) Definiteness. Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
- 3) Input. An algorithm has zero or more inputs, i.e. quantities which are given to it initially before the algorithm begins.
- 4) Output. An algorithm has one or more outputs, i.e. quantities which have a specified relation to the inputs.
- 5) Effectiveness. An algorithm is generally expected to be effective. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in finite length of time by a man using pencil and paper.

In Computational Complexity Theory computation, algorithms, and functions are studied. If an algorithm exists, it can be implemented on a Turing machine:

Christos H. Papadimitriou "Computational Complexity":

"Despite its weak and clumsy appearance, the Turing machine can simulate arbitrary algorithms with inconsequential loss of efficiency. It will be our formal model for algorithms in this book." [130 Ch 2 p 19]

We see that, adding these different sources, some agreement can be found. The clue to understanding the algorithm concept lies in that the algorithm must be separated from its execution. Every possible way of performing an calculation do not necessarily imply that an algorithm exists describing that calculation.

Looking at a computation, the algorithm is its corresponding description. The execution of the computation, the actual calculation, then is an interpretation of the algorithm. This enable us to concisely define an algorithm.

The execution of the algorithm must be a computable task (computability), so that we are guaranteed that a computer will perform a finite calculation sequence and then stop. In addition to that, there must be a knowledge, so that we may be able to tell, in advance, that if the calculation corresponding to the algorithm is performed, then the calculation will stop, with the intended result value.

The algorithm will be defined as a Turing machine property. The well-known Turing machine definitions are given below:

Definition 1. The Turing Machine Predicate.

By $T(z,x,y)$ we mean that the Turing Machine z when started with x as input, produces a computation sequence (output) y . $T(z)$ is the machine z when the input x is unspecified. If $T(z,x,y)$ terminates with a finite computation sequence y we say that $T(z,x,y)$ is TRUE. If this is not the case $T(z,x,y)$ is FALSE, and the machine will compute forever. Note: We may replace the cumbersome output y by some suitable output channel where the machine can write "TRUE", or whatever is convenient.

Note: This definition refers to a "stopping" machine. There are also "generating" Turing machines, that always compute forever. A stopping Turing machine corresponds to a halting algorithm like a sorting algorithm. When a list is sorted, the sort algorithm terminates. A generating Turing machine could correspond to an iterative algorithm. The purpose of the iterative algorithm is for each iteration to improve a solution to a problem. In principle the calculation could go on forever, but in practice it is stopped using external means, when sufficient precision has been obtained.

Definition 2. The Universal Turing Machine

A Turing Machine $U(z,x,y)$ is universal if its instruction set is sufficiently rich to allow it to simulate any other Turing Machine.

Such a simulation will normally not be slower than the original computation, except by a constant factor. By definition 2 we mean a "direct" simulation. By contrast we may think of two different machines, computing two similar functions, which eventually yield the same result.

The input x to a Universal Machine $U(z)$ may be split into $x=\{P_x,D_x\}$ where P_x is the program, that we wish to execute, and D_x is the input data to that program.

Definition 3. The Stored Program Computer.

By a Stored Program Computer= $U(z,P_x,D_x,y)$ is understood a Universal Turing Machine z , which have the program P_x included with the input data D_x in $x=\{P_x,D_x\}$, and not built in into the machine z .

This is equivalent to the Universal Machine, Definition 2. [141 section 7] A "Stored Program Computer" is also called a "microprocessor" if the software/memory is externally connected to the microprocessor silicon chip. If the software/memory is included in the same casing, most often on the same chip, the computer is called a "microcontroller".

Definition 4. An Algorithm.

A computer program P_x for a Universal Turing Machine $U(z,P_x,D_x,y)$ is an algorithm, if and only if we have an advance knowledge about what the corresponding execution $U(z,x,y)=U(z,P_x,D_x,y)$ will perform, and that the execution will halt with a finite calculation sequence y .

This definition reflects the fact that the algorithm of $U(z,P_x,D_x,y)$ is the program P_x . It is the input program P_x , and not the behaviour y , observed when executing (P_x,D_x) , that is the algorithm. The essential property of an algorithm-program is that we have a knowledge: we know what will happen when we execute an algorithm-program [143 ch 3.1 p 23]. We also know that the machine $U(z)$ given the algorithm P_x will halt, with $U(z,P_x,D_x,y)=\text{TRUE}$. The defined algorithm concept agrees with the previous given definitions of an algorithm; both when used in a complexity theory setting and when used in computer science.

That Definition 4 opens up the possibility to construct a cipher system that is considerably different, compared to ordinary algorithms, may not be at all evident. Classical encryption algorithms, like the German Enigma machine, the DES, or the AES, seems to be examples of a general approach to the

construction of encryption systems. All these systems may be described by an algorithm P_x both in a software implementation or in a hardware implementation in VHDL. To reach beyond the algorithm-model of encryption systems, we must make a change in the model, that we use for the construction of the encryption system. The change that is needed is simply to stack two algorithm-running computers on top of each other.

The Properties of the Interpreter

A software or a machine that scan a structured input, and build a data-structure describing the input is called a Parser. If the Parser output a translation of the input in another language we call the software a Compiler. A compiler normally read an input software in a high-level language like C, Pascal, ADA, or COBOL; or VHDL or Verilog for a hardware design, and then output a functional equivalent in a low-level language like P-code, hardware assembly, or three-address-code; or a RTL hardware description. An Interpreter perform the functions of a Compiler, a translation, but when a Compiler output a low level language equivalent, an Interpreter go ahead and execute these instructions directly. BASIC is a well known example of a language intended for interpretation, but in principle any computer language can be interpreted.

We continue with a study of the properties of the Interpreter. The Interpreter will be our model for an implementation of the HCIA cipher. There are many kinds of interpreters. For the HCIA we will need an interpreter that has a general (Universal) language like COBOL, C, ADA, BASIC, VHDL, ...

The Interpreter.

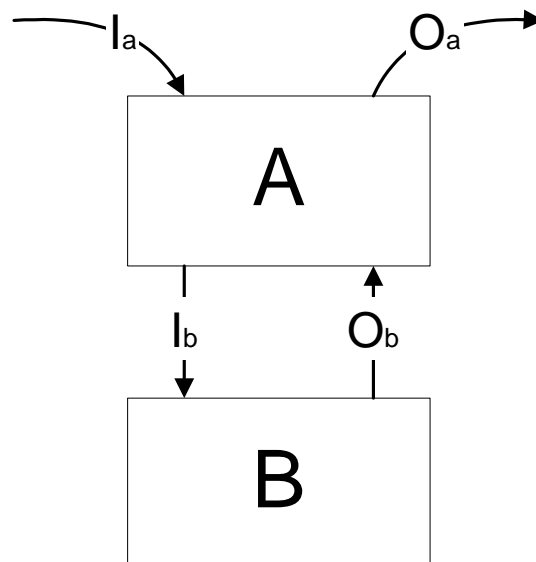


Figure 6. Interpreter.

An interpreter could consist of two connected machines, A and B. Machine A has an input channel I_a and an output channel O_a . Connected to the machine A is a Machine B, that is universal. The input to Machine B, I_b , comes from Machine A. The output from machine B (O_b) is feed into machine A, which may in turn output something on the output channel O_a . Machine A is executing a program

$P_x(A)$, that translates the input I_a to I_b and the output O_b to O_a . Machine A is also allowed to do other processing. We assume that the program for the translation, $P_x(A)$, is an algorithm according to Definition 4.

First, we may run ordinary algorithms on the machine. There must exist programs $P_x(A)$ where the output O_a of the interpreter, given any input I_a , may be simulated by another machine C executing a program $P_x(C)$ which is an algorithm. To reach this all we have to do is not connect Machine A and B in picture 2, and give machine A a suitable algorithm-program $P_x(A)$. But there exist an interpreter program $P_x(A)$ where the behaviour of the interpreter is the same as for a universal machine U. This is constructed by connecting the A and the B machine. When machine A run an interpreter program $P_x(A)$ and is given an input I_a this input is transferred to the B machine in the form of a concatenation of an input program $P_x(B)$ and a data input $D_x(B)$. The machine B then compute the function defined by $P_x(B)$ with the argument $D_x(B)$. The output of machine B is then transferred to the output of machine A. If the computation defined by $P_x(B)$ is computable for the argument $D_x(B)$ the machine B will write "TRUE", and if not it will compute forever.

We see that a Stored Program Computer (Definition 3) or a Universal Turing Machine (Definition 2) has been implemented.

It is now easy to see why an interpreter is no algorithm: If the interpreter (picture) is an algorithm we would know, for all inputs I_a , what would happen if the interpreter would interpret the input string I_a (Definition 4). But because the string I_a merely contain a program and an input for the machine B, we cannot know anything about what the interpreter will do, because we don't know what machine B will do. We conclude that an interpreter cannot be an algorithm.

We now look into this in more detail, and identify the non-algorithmic part of the interpreter:

We see that an interpreter cannot be an algorithm, if we look at the interpreter from the outside, as a black-box. The interpreter itself, on the other hand, can be implemented as an algorithm. Suppose that we choose Basic as our interpreter. The Basic interpreter is an algorithm that read an input file, perform syntax checking, and then start executing the program from the first source line. The execution of each expression and each statement can be precisely defined. Each such statement can be implemented as a sub-algorithm. The complete interpreter would consist of a collection of such algorithms. As the interpreter only consists of a collection of well defined algorithms, the complete interpreter will be an algorithm.

When we look at the interpreter from the outside, we normally merely consider the interpreter as a translation tool. The algorithm in question, written in Basic, will be given to the interpreter. We say that it is the Basic program that output symbols, silently ignoring that it is really the interpreter that is doing this. We say that there is a syntax error in the Basic program, when the interpreter output this message. We say that there is a bug in the Basic program when the interpreter loops forever.

That the interpreter cannot be any algorithm is equivalent to that we cannot predict the properties of the output. This occurs whenever we observe the interpreter itself and its output, given some arbitrary input. Suppose that you where guessing at the output from a Basic interpreter given an unspecified input. Impossible! The input, the input Basic program, defines the output in all conceivable ways. The program may run in loops, that may break on very delicate and complex conditions. The program may be small or large. The program may, during execution, perform any calculation or run any (sub-) algorithm. It may be another interpreter, for any language, as long as the interpreter itself is written in Basic. It may run one or several tasks simultaneously.

That no essential property of the output of the interpreter can be obtained by analysis of the interpreter itself is, in this setting, a very obvious result. The properties of the interpreter necessary for this scenario is:

- (A) The language interpreted by the interpreter must be a general language, so that the machine simulated by the interpreter is an Universal Machine.
- (B) The input must be unknown to the external observer.
- (C) The output from the interpreted software must influence the output from the interpreter sufficiently.

To create a cipher based upon an interpreter, we must be careful not to remove these three properties. We may, however, add properties that we need especially for our cipher implementation:

- (D) There should not be any input syntax checking. Rather, any binary string should be allowed as input to the interpreter. This possibly looks confusing or difficult. How do I transform random binary data into an encryption algorithm? But it is easy, as will be shown below.
- (E) We will need a steady stream of output from the interpreter, independent of the software executed. This can be accomplished by arranging so that the "Machine B" occasionally output symbols. It is important (or an advantage) that the exact time, when a symbol is output, is influenced by the running software.

Implementation Issues

The interpreter, a valuable model for analysis, is however not the model that was selected for the example implementation. Rather we will use a microprocessor model instead. This also have the advantage that a lot more readers have previous knowledge about how a computer works compared to how a parser/compiler/interpreter is internally constructed.

A detailed description on how a HCIA system may be constructed can be found in [163]. When reading [163] kindly remember that, as this is a patent application, the text often suggest that a task may possibly "in an advantageous embodiment of the invention" be implemented in a certain way. The easiest way of obtaining a solution that works is to include (accept) all these suggestions.

The implementation of [163] also exist as a C++ software. Please contact Protego Information to obtain a copy of the software.

This text is about the use of the HCIA for encryption. The example implementation is about constructing a pseudo-random generator to mask statistical deficiencies in the raw output from a hardware random number generator. This example have the advantage that it is not as complex as an encryption implementation, and it show how the described technology can be adopted and adjusted to to reach the target goals of the implementation. So now we will be discussing TRNG:s for a while!

Implementation Example: The R230 TRNG

A pseudo-random number generator (PRNG) is a deterministic procedure that map a "small" set of input symbols onto a countable set of output symbols, that are output one at a time by running the PRNG in a loop. A true random number generator includes a hardware random number source that make the output unpredictable in the information-theoretic sense (the output can not be approximated or predicted by any computing device or method).

Below we shall build a HCIA system for processing of random numbers. The background is that, due to limitations of hardware, the raw input of random numbers from a random number source may have correlations and bias, that is undesirable in most application areas. At first glance it may be thought that correcting these statistical problems may be wrong, as the output will be a combination of a deterministic source and a hardware source. But if we take a measurement of the amount of information that exist in a block of input from the hardware random number source -- we ignore measurement problems for the moment -- then it would not be wrong to output this amount of information from the TRNG. Preferably the information shall be encoded (compacted or compressed) onto a statistical bias-free output symbol stream.

In practice a conservative estimation is made about the input information rate of the input stream, and the data is "compressed" by mixing a pool of input symbols using some kind of encryption system, a shift register, or similar arrangement. Information is then output from the pool at a rate lower than the rate that information is (estimated to be) input to the pool.

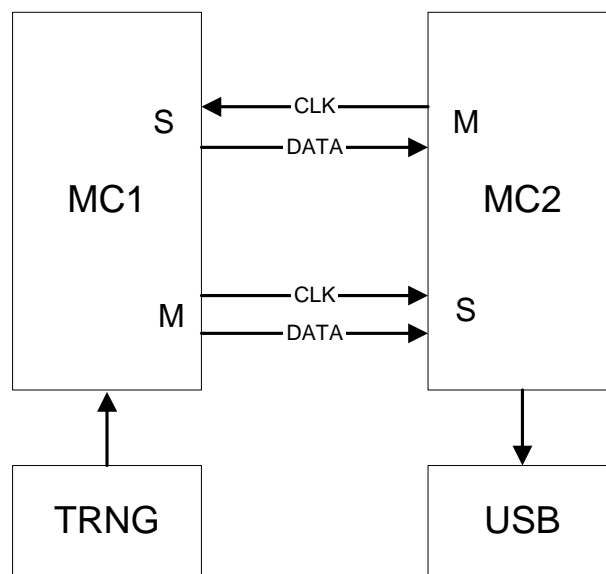


Figure 7. HCIA experiment setup.

Experiment Setup

The application use two microcontrollers where a first microcontroller is attached to the random number source and a second microcontroller is attached to a communication circuit (USB line driver).

The two MC:s communicate using a communication link, consisting of a SPI-Master \leftrightarrow SPI-Slave pair (Serial Peripheral Interface [165]). The implementation have two independent communication links, with independent clocks. Each MC have a Slave unit and a Master unit.

The first MC take a number from the random number source and process it through the HCIA system. It then write the result to a SPI-Master unit, that in response to this starts and send the number into a SPI-Slave unit in the second MC.

The First MC then obtains a second byte from the hardware and then wait for the SPI-Slave unit, on the other link, to become ready. When ready the first MC send an unprocessed byte to the second MC.

The second MC waits for the slave-unit to have an input. The Slave unit is connected to the Master unit in the other MC, so the data received will consist of a HCIA processed byte. This will be output to the USB communication unit (to the receiving computer). Then, the second MC drop a dummy byte into its SPI-Master, to commence clocking of the slave unit in the first MC. The previous byte received from the first MC is then obtained, and then processed by HCIA, and sent to the USB communication unit.

This construction with Master-Slave units enable both MC:s to stay synchronised with each other. The communication time delay and the one-byte buffer of the SPI units is used as a communication buffer.

The highest possible output speed shall now be obtained. This is limited to 1.0 MB/s by the USB bus. (MB = MBytes; Mb=Mbits). The raw speed of the noise source of the R230 unit is about 12Mb/s=1.5MB/s. We evidently have enough input and output capacity. What we lack here is sufficient processing power: each MC is capable of 24Mclocks/s. We have (if we add and count both MC:s) 4 clocks/input bit or 6 clocks/output buffer (capacity) bit.

Since we try to adopt a cipher solution to a real application, detailed knowledge of the application is needed to understand this example HCIA implementation. If we had to adopt the other way around it would be rather strait-forward to describe the cipher; compare with the DES!

The MC:s have a rather significant drawback: they have only 256 bytes memory. Yes, 256 bytes or 0.000244 Megabytes of memory! This is clearly a significant drawback for the HCIA cipher, that desperately need a large memory, to be able to simulate "general" or "universal" computation. But for sake of exercise we shall try our best to make a solution using available resources!

We cannot use all 256 bytes to the HCIA system. The memory shall also hold application-specific variables and buffers as well as the CPU stack.

We aim for an 128 bytes HCIA memory. There might be a few extra bytes used. The memory model for the HCIA implementation is random-access memory with a fixed size. Compare with the implementation in [163] where three stacks are used to implement the memory.

Cycles Budget:

The following table show how many clock cycles we may use totally when processing a byte to reach certain speeds:

Output 512 kB/s = 4096 kb/s:	91.5 clocks
Output 256 kB/s = 2048 kb/s:	183.1 clocks
Output 200 kB/s = 1600 kb/s:	234.3 clocks
Output 128 kB/s = 1024 kb/s:	366.2 clocks

For the first MC the noise sampling and communication loop consumes a minimum of 64 clock cycles. For the second MC the I/O loop consumes 114 clock cycles. In our application we will process only random numbers. If we remove the "USB buffer full"-checks we can speed up the second MC to 79 clocks. We see that the second MC is the time-limiting MC with 79 clocks/loop.

Available encryption time as a function of transfer speed:

```
Output 512 kB/s = 4096 kb/s:    12.6 clocks
Output 256 kB/s = 2048 kb/s:   104.1 clocks
Output 200 kB/s = 1600 kb/s:   155.4 clocks
Output 128 kB/s = 1024 kb/s:   287.2 clocks
```

Example: $(2 \cdot 24E06 \text{ clocks/s}) / (128 \cdot 1024 \text{ bytes/s}) = 366.21$ The factor of "2" comes from that two MC will share the work. Subtract 79 to get the 287.2 in the table above.

The HCIA subsystem works like the computing engine of a microcontroller. It has a memory, registers, and some operations that perform update operations on the registers, the memory, or both.

In the example application the input byte consist of a byte of raw randomness from the hardware source. The hardware have a compensating circuit that eliminate input bias almost completely. The correlation between bits is also very low. This means that we can, for now, assume that the input will be a random byte with almost no statistical deficiency.

The HCIA system is given an input byte, taken from the hardware. The first task is to calculate an operation number in response to this input. Preferably shall we also update some registers (or memory) directly with the input. Such an update, that is performed with all input bytes, force the memory contents of the updated memory to be a function of all input.

Or, more structured:

- 1) The first phase in an HCIA encryption round is to mix the input with the machinery.
- 2) Then shall an operation be selected. The operation will conduct some update on the assigned registers and memory. A single or preferably a series of operations shall be executed.
- 3) Finally some data from the computation is collected as the output from the HCIA subsystem.

As an example of how the HCIA system may be adjusted to accommodate different needs, the following notes was saved during the construction of the example HCIA system.

Notes on Preliminary Versions

Given one byte of input, the information in this byte should be distributed to the HCIA system, to the registers and/or to the HCIA memory. Then we shall select an operation based upon the result of the update, or during the update operation.

The input byte is held in register "A". The first attempt to distribute information from the "A" register to the memory was:

```
ADD  A, (D1)  6
MOV  (D1),A  5
INDEX table 13 // A=function(A)
ADD  A, (D2)  6
MOV  (D2),A  5
```

For a detailed description of the assembly code instructions, see [164].

Further, the best way to select an operation is:

```
AND  A, 0x7E  4
JACC baseaddr 7
```

```
baseaddr:
<list of JMP instructions>
```

Selecting an operation takes 16 clocks, using this technique. The processing above take 35 clock cycles. This makes 51 clock cycles. This is too much and must be reduced.

The X register for indirect addressing [164 ch 3.3.3] may be used to implement a memory model for the HCIA. Suppose that we assign the "X" register a value of 0..63. The address X+base will then point to a byte in the lower half of the memory and X+base+64 will point to a byte in the high half of the HCIA memory. This addressing enable an operation to access two bytes using the same X index. We call these addresses XLow and XHigh.

The second approach (of the mixing function) was to update using the X indirect register:

```
XOR (XHigh), A 8
ADD A, (XLow) 7
ADC (reg), A 7
MOV A, (reg) 5
Do operation 16
```

This is also too slow! Note that for every selected operation these lines will be executed. This is a good but stereotype update.

The important point is that running two or several smaller operations generate much more complexity than just one complex update. In the intended application area we may also depend on that the input is almost random. Based upon these assumptions, we chose not to update the "A"-register before the operation is selected. We rely upon that the input "A" is always a "good" value. This we must fix outside the HCIA update loop.

We take effort to save all input bits, though, by adding the input "A" to a register, before we convert the number to an operations number.

```
ADD (register), A
Do operation:
    AND A, 0x7E ; value is byte index, 64 indexes, even numbers
    JACC Base_address
Base_address:
    JMP Operation_00
    JMP Operation_01
    ....
    JMP Operation_63
```

HCIA Data Structure for the Example Implementation

Reg_In	1 byte	Contain all information available in "A" when processing was started.
Registers	4 bytes	The "CPU" registers.
Memory	128 bytes	Fixed address addressing and "X" register relative addressing of two bytes.

Notes on The Operations

The recommendation would be to iterate HCIA-operations a while before exiting. But we are short of cycles, so the following is proposed: 50% operations exit immediately and 50% iterate if a test bit is set. When exiting shall the operation provide an output, in register "A", consisting of a linear combination of the "Reg_In" register and data from the HCIA memory.

X register:

The X register shall be updated or set in about 20%-40% of available operations.

Move operations:

Shall move information from memory to registers or from registers and memory.

Compute operations:

Shall compute a result to be put into a register or an update of the "X"-register.

Other operations:

Special or more complex operations.

Loop Warning: It is possible for the machinery to get stuck in a loop. We need some external means to break such loops.

Every implementation is a compromise between different goals. To simplify the implementation we choose to move all update loops outside the HCIA-machinery. Second, as random bytes will be processed it makes sense to load new randomness into the HCIA instead of reusing old data. In an encryption setting it would be more advantageous to iterate a few times on old data.

The Final Version

For the example implementation -- where we know that the input will have good randomness properties -- we exclude most of the alternatives of update, that we may perform with the input, in the operation-selection phase. It would be more efficient to use available clock cycles in the operations themselves instead, as each operation can perform the update slightly differently.

This is the start up sequence:

```
HCIA_Processing:
    ADD    [ Reg_In], A           // 7
    AND    A, 0x7E               // 4
    JACC   Operation_table      // 7
```

- 1) The A register contain the input byte, in our application a random number with 99.8% randomness. The input is in "A" register.
- 2) We add the input to the memory cell [Reg_In]. This is to save the information in the input for later use.
- 3) We let the input define the operation number directly. This is normally a bad thing. Simply assume encryption of the string 000...000. Then will only the operation "00" be executed. Note how easy this is to avoid once we have observed this simple fact. For the intended implementation this is not a problem and consequently it does not have to be solved.
- 4) We mask out six bits by setting the MSB and the LSB bits to zero. The number is now an even number in range 0..126.
- 5) The JACC instruction jump to the address that is the sum of the "A" register and the supplied address.
- 6) The resulting address is another jump instruction in a jump table. Each of the 64 possible inputs correspond to a JMP instruction. We see that "A" must be even, as the JMP instructions are two bytes long.

This is the operation selection table:

```
Operation_table:
    JMP Operation_00
    JMP Operation_01
    JMP Operation_02
    JMP Operation_03
    JMP Operation_04
    JMP Operation_05
***
    JMP Operation_60
    JMP Operation_61
    JMP Operation_62
    JMP Operation_63
```

So, as an example, if A=0x7E "JMP Operation_63" will be executed and if A is zero "JMP Operation_00" will be executed.

This operation, at the heart of the HCIA subsystem, is an implementation of a function selector. Its implementation is different in different languages; in C we could use an array with subroutine addresses.

We now comes to the most interesting part of the HCIA system: The operations themselves. Before we begin remember the essential properties of the HCIA function:

- Should work similar to a microprocessor.
- Shall have a large memory.
- The operation set should be rich enough to enable full computability.

"Full computability" is easy to achieve. We merely need to meet a few very simple demands. How do a microcontroller or microprocessor works? What is it that enable us to solve any computational problem as a software?

- 1) We need a "register" or "register bank" to work with.
- 2) There should be operations that transfer data from the registers to the memory, from the memory to the registers, possibly also between memory locations.
- 3) There should be some operations that perform calculations using the registers as source and destination.
- 4) Normally there should also be a control structure that enable implementation of loops and jumps in the software.

Notes:

(1): It is advantageous to implement a rather large register bank. In this implementation the register bank is four bytes. Note that the target MC have only a single one-byte accumulator.

(2) and (3): The requirement is only that there shall exist operations that fulfils this requirement. If a subset of the instructions enable full computability, then this will be the case for any superset.

(4) can be excluded in our particular example implementation, as the computation sequence will be defined using a hardware random number source. Due to this, for this particular application area, it is more advantageous to take a new fresh byte of randomness, compared to reusing old input bytes. In an encryption application it would be advantageous to also implement a loop structure.

A real microcontroller use simple atomic instructions. This is to make the MC easy to use. This will also minimise the hardware size of the microcontroller implementation (chip area). Using an orthogonal instruction set, preferably a complete orthogonal instruction set, is an important requirement for modern microcontrollers.

But we shall implement a cipher, so these requirements to not apply to us! We need maximum complexity on minimum clock cycles. Therefore we shall enforce that each HCIA-operation have obscure and complex side effects. In particular shall we mix linear operations (MC addition, subtraction, and exclusive-or) with nonlinear operations like table-lookup, binary AND, and binary OR. In case that the target computer have a hardware multiplication unit, so that multiplication is an effective operation, we can also include binary multiplications in the HCIA-operations. The complexity of atomic operations has previously been studied in [134]. See also [145] (in English).

Side-effects can be obtained by updating locations in the memory or registers. The HCIA-operations shall also, generally, read information from several locations and update information at several locations.

Note that we do not limit the computational capability of the HCIA by adding side-effects to the operations in the HCIA operation list. It is merely that we cannot comprehend how the input symbols shall be encoded to make the HCIA-style-computer perform a well-defined task, such as sorting a list of numbers or adding two numbers forming a sum, and so on.

Now back to our example-implementation:

When we start each operation, in the operation list, the encoded operation number is held in the "A"-register. This value is a constant, for each operation, as the operation is selected based upon the contents of the "A" register. Consequently shall this number be discarded.

We demand of all other software tasks (all other software+interrupt routines) that they must preserve the "X" indirect register. This make the "X" register available to the HCIA subsystem. "X" is loaded with an integer 0..63. This enable the "X" register to indirectly access two legal memory addresses: [X+XLow] and [X+XHigh] defined as [X+HCIA_RAM + 0] and [X+HCIA_RAM + 64]. In the code the mnemonics [X_Low] and [X_High] are used for indirect memory references.

Now we define the HCIA-operations:

```
// Operations read from memory and update registers

Operation_00:          // Numbers to the right are instruction clock cycles!
// =====
// Op 00 Clocks:  56
MOV  A, [ X_Low ]      // 6
ADD  A, [ Reg_2]       // 6
INDEX Subst_Table_0   // 13
RLC  A                 // 4
ADC  A, [ Reg_3]       // 6
MOV  [ Reg_2], A      // 5
XOR  [ X_High], A     // 8
RET                       // 8
```

Operation 00 start by loading A with data from the memory [X_Low]. This is an indirect reference, so it will take 6 clock cycles. Then we do a bit of computation: We add [Reg_2] to "A", forming an intermediate result. Then we perform a table look-up. The example implementation have two 256-row 8 bit substitution tables. The MC have special instructions that do a table look-up. We use two such tables: Subst_Table_0 and Subst_Table_1. These will be randomly selected substitutions. In the substitutions each byte value occur at exactly one index value. After the substitution we rotate the "A" register left one bit through carry. The input carry bit is the MSB carry bit from the previous addition. We then add [Reg_3] and the new carry bit to the "A" register. We then move the resulting "A" register to the [Reg_2] register. Finally we XOR the "A" register with the [X_High] memory cell; result stored in [X_High]. The operation return with the result obtained in the "A" register.

We see the typical pattern: We load data from memory and a register, perform a calculation, and store the calculation result in a register, memory, or preferably both. We also see a transfer of information from [X_Low] to [X_High] memory cells. Note that, when the [Reg_2] is overwritten, this is no problem as the information contained in [Reg_2] was input to the calculation in a previous stage.

In this operation the input carry to the operation is not defined. But the carry flag is always defined due to the addition instruction that is performed prior to the selection of operations.

See how easy we can cryptanalyse the HCIA system -- we merely need to know the selected operation number, and previous memory and register contents!

Next operation:

```
Operation_01:
// =====
// Op 01 Clocks: 43
MOV  A, [ X_High]           // 6
ADD  A, [ Reg_3]           // 6
RRC  A                      // 4
MOV  [ Reg_3], A           // 5
ADC  A, [ Reg_2]           // 6
ADC  [ X_Low ], A          // 8
RET                           // 8
```

The operation 01 is a little different in its structure. There is no table lookup. The operation read input information from [X_High], [Reg_3], and [Reg_2]. The operation update [Reg_3] and [X_Low]. Most of the contained calculations seem to be linear, and the operation also include a bit rotation. Note that the registers selected are not essential -- you may, without introducing weaknesses, modify this if you wish. The important thing is that there are several operations that read the memory and update the registers.

Next operation ...

```
Operation_02:
// =====
// Op 02 Clocks: 43
MOV  A, [ X_Low ]          // 6
ADD  A, [ Reg_0]           // 6
RRC  A                      // 4
MOV  [ Reg_0], A           // 5
ADC  A, [ Reg_3]           // 6
ADC  [ X_High], A          // 8
RET                           // 8
```

This operation is almost identical to operation 01. We see that the only difference is that other registers has been selected.

// Operations read from registers and update memory

```
Operation_06:
// =====
// Op 06 Clocks: 54
MOV  A, [ Reg_0]           // 5
XOR  A, [ Reg_1]           // 6
ADD  [ X_High], A          // 8
INDEX Subst_Table_1        // 13
ADC  A, [ Reg_In]          // 6
ADD  [ X_Low ], A          // 8
RET                           // 8
```

We also need operations that read from the four registers [Reg_0], [Reg_1], [Reg_2], and [Reg_3] and update the memory at [X_Low] and [X_High]. The above operation is an example of this style of operations.

Operation 14 contain code for updating the X indirect register. This enable operations to access different parts of the memory:

```
Operation_14:
// =====
// Op 14 Clocks: 55
MOV  A, [ Reg_2]           // 5
AND  A, HCIA_RAM_Mask     // 4
SWAP A, X                 // 5
OR   A, [ Reg_3]         // 6
INDEX Subst_Table_0      // 13
XOR  A, [ X_Low ]       // 7
ADD  [ Reg_2], A         // 7
RET                               // 8
```

We see that the six bits that point to a memory location is taken from [Reg_2]. The constant "HCIA_RAM_Mask" is included in the source code to make it easy to use this particular HCIA-implementation also with other memory sizes, should the need arise. By storing the new index value into "X" with the SWAP instruction, may the old "X" value influence the output. This cost only one extra clock cycle.

```
// Exchange operations

Operation_16:
// =====
// Op 16 Clocks: 48
MOV  A, [ Reg_In]        // 5
SWAP A, [ Reg_3]        // 7
SWAP A, [ Reg_0]        // 7
SWAP A, [ Reg_1]        // 7
ADD  [ X_High], A       // 8
XOR  A, [ Reg_3]        // 6
RET                               // 8
```

There are also some operations that move information around. But, it doesn't hurt with an ADD or XOR, if this can be included without a high clock-cycle penalty.

We then have more complex operations. We may include, in addition to simple elementary operations, any operations that do any kind of update. The full computability set is complete -- we can never escape from this set -- so we have a considerably freedom when the HCIA-operations are constructed.

To prevent a more complex update from slowing down the whole system, we can perform more lengthy updates only occasionally, as in the operation 33 below:

```
Operation_33:
// =====
// Op 33 Clocks: (1/2)*31 + (1/4)*53 + (1/4)*110 = 56.25
MOV  A, [ X_High]       // 6
ADD  A, [ Reg_1]       // 6
JC   Op_33_1           // 5

ADD  [ Reg_0], A       // 7
RRC  A                 // 4
ADC  A, [ Reg_3]       // 6
JC   Op_33_1           // 5
```

```

XOR    A, [ Reg_2]           // 6
INDEX  Subst_Table_1       // 13
ADD    [ Reg_0], A         // 7
RLC    A                   // 4
RLC    A                   // 4
ADD    [ X_Low ], A       // 8

AND    A, HCIA_RAM_Mask    // 4
SWAP   A, X                // 5
OR     A, [ Reg_0]        // 6

Op_33_1:  XOR    A, [ Reg_In] // 6
          RET    // 8

```

The operation fork at two places, using the "JC" "jump if carry"-instruction. The probability that the carry will be "1" can be estimated to be about fifty percent. This yield a 100%-50%-25% distribution for the three instruction sequences.

In fact this expands the effective instruction set for the HCIA implementation. One kind of update will be performed for some inputs, other updates for other inputs. This behaviour is also sometimes observed in real microprocessors, most often sending their programmer into despair. Note in particular that the the index register is updated with 25% relative frequency.

We end this section, with operations and notes, with a nasty update! This update is triggered when a computed sum (byte) is zero.

```

Operation_34:
// =====
// Op 34 Clocks: 255/256*44 + 1/256*2280 = 52.73
MOV    A, [ Reg_3]         // 5
AND    A, [ Reg_In]       // 6
SUB    A, [ Reg_1]        // 6
JZ     Op_34_1             // 5   JMP (1/256)
ADD    [ X_High], A       // 8
MOV    A, [ X_High]       // 6
RET    // 8

Op_34_1:  // Put index reg at random pos
          // A is zero
MOV    X, 35              // 4
MOV    A, [ Reg_0]       // 5
AND    A, HCIA_RAM_MaxINX // 4
ADD    A, HCIA_RAM       // 4
MOV    [ MVI_A], A       // 5
MOV    [ MVI_B], A       // 5

Op_34_2:  // Loop: Average 35 * 65 = 2275 clocks
MVI    A, [ MVI_A]       // 10   Memory end: return zero
JZ     Op_34_3           // 5
ADD    [ Reg_3], A       // 7
MOV    A, [ Reg_3]       // 5
XOR    A, [ Reg_In]      // 6
INDEX  Subst_Table_1     // 13
MVI    [ MVI_B], A       // 10
DEC    X                 // 4
JNZ    Op_34_2           // 5

Op_34_3:  XOR    A, [ Reg_1] // 6
          ADD    A, [ Reg_3] // 6
          AND    A, HCIA_RAM_Mask // 4
          MOV    X, A // 4
          MOV    A, [ X_Low ] // 6
          ADD    A, [ Reg_In] // 5
          RET    // 8

```


When the tests show an acceptable output bias level -- still much too high to allow analysis using any other statistical tool than the 0/1 bias test -- we go to the next step. To remove the annoying 0/1 bias, we include a linear component to the random number processing subsystem. The linear component should preferably use "a large" buffer memory. The recommendation would be about 10.000 bytes, but for this MC we will have to do with a handful of bytes.

So, five buffer bytes [Reg_Y1]..[Reg_Y5] is allocated. The output from the HCIA system is added to these registers, that are used one at a time. This function clearly does wonder to the 0/1 bias problem, eliminating this statistical problem from the output.

An example of an update operation would be:

```
CALL HCIA_Processing
ADD  A, [ Reg_Y1]
MOV  [ Reg_Y1], A
```

The "A" register is output to statistical testing and is then used as the next input to the HCIA function

```
CALL HCIA_Processing
ADD  A, [ Reg_Y2]
MOV  [ Reg_Y2], A
```

... and so on for five registers. At the beginning of the test loop is a counter added to the "A" register, as previously explained.

There are a few statistical problems that remain. This is due to that the tests detect the five-byte cycle, and detect a very small but measurable correlation between each fifth byte. This is eliminated using the following modifications.

- 1) We increase the number of feedback registers from five to eight.
- 2) We change the atomic additions above to a carry-add chain. This is implemented by storing the carry flag from the previous addition and incrementing the result, of the next addition, in case the carry flag was set. This implies that the buffer registers is implemented as a single large integer, rather than several small eight-bit integers.
- 3) We make use of a trick and extract the carry bit, of the HCIA-operation, after the seventh iteration. As previously explained, the value of the carry flag is set, and is valid, after each HCIA-operation. Now, if the flag is set, we skip updating of the eight register, and move directly to the update of the first register.

The result of the above update operations is that the cycle length, of the linear buffer, is randomly varying. This trick de-synchronise the statistical tests, and bias and correlation can no longer be detected. The intra-byte correlation is removed by letting the carry bit transfer information from one bit to the next, through all bits in a byte and through one byte to the next, and finally from the highest byte back to the beginning of the first byte of the linear buffer.

The software was tested with the Crypt-X statistical test suite for PRNG:s and key generators [166]. The implementation was also tested with the NIST statistical test suite. These two statistical test packages are the two most potent test packages available for statistical testing of encryption functions and PRNG:s (stream ciphers) today.

A software construction issue, of no importance, is that the addition chain is duplicated in the source code: one chain for "normal" update and one for "carry=1" update. This is due to that the carry flag is saved by jumping to another program location. (Not elegant, but fast!)

The purpose of the HCIA system, in the TRNG example implementation, is to introduce a non-linear component into the output data. That the statistical tests can be overcome by a more or less clever linear feedback, is evident from the discussion above. The non-linear component is needed to make sure that no application or statistical test intentionally, or by accident, includes a structure that "removes" the effect of the linear operation. The nonlinear properties of the HCIA-operations may also be tested using the statistical test suites. These tests make clear that the HCIA-operations are extremely nonlinear, as is to be expected. A note, however, is that the HCIA memory should not be reset to zero. It is necessary, for the correct function of the operations, that most locations are non-zero. In the example implementation this can easily be arranged by loading the HCIA memory with randomness from the hardware random number generator at start-up.

Above a description of some operations is made in detail. Below we include the entire HCIA source code. The purpose with this is NOT to encourage anyone to scrutinise the source code. Instead the operations shall be viewed briefly, with the aim of looking at the forest and not getting stuck at a particular tree. At time of writing, the operations remain preliminary. Please contact Protego Information in case you need to implement this example into a product.

Source Code

```

HCIA_RAM: EQU 127 // 129 regs
HCIA_Reg: EQU 112 // 15 regs

HCIA_RAM_Mask: EQU 63 // 63 <-> 128 byte RAM; 31 <-> 64 byte RAM
HCIA_RAM_MaxINX: EQU 127 // Size-1

MVI_A: EQU HCIA_Reg + 0
MVI_B: EQU HCIA_Reg + 1
Reg_In: EQU HCIA_Reg + 2
Reg_Y1: EQU HCIA_Reg + 3
Reg_Y2: EQU HCIA_Reg + 4
Reg_Y3: EQU HCIA_Reg + 5
Reg_Y4: EQU HCIA_Reg + 6
Reg_Y5: EQU HCIA_Reg + 7
Reg_Y6: EQU HCIA_Reg + 8
Reg_Y7: EQU HCIA_Reg + 9
Reg_Y8: EQU HCIA_Reg + 10
Reg_0: EQU HCIA_Reg + 11
Reg_1: EQU HCIA_Reg + 12
Reg_2: EQU HCIA_Reg + 13
Reg_3: EQU HCIA_Reg + 14

XLow: EQU HCIA_RAM + 0
XHigh: EQU HCIA_RAM + HCIA_RAM_Mask + 1 // Half Size:
// HCIA_RAM + HCIA_RAM_Mask
Stoppcode: EQU HCIA_RAM + HCIA_RAM_MaxINX + 1 // Memory Size:
// HCIA_RAM_MaxINX + 1
// Total size including
// stopp code: 1+Memory Size

#define X_Low X+XLow
#define X_High X+XHigh

R300_startup:
    INC [WD_Counter] // 7 Trigger resetcontrol
    Set_Command_Idle
    CALL Set_CMD_TX8
    CALL HCIA_Setup
    JMP R300_loop_N

```

```

Skip_C:
    INC    A
    ADD    A, [ Reg_Y7]
    MOV    [ Reg_Y7], A

    JC    Loop_Y1C
//      JMP    Loop_Y1

Loop_Y1:
R300_loop_N:
    INC    [ WD_Counter]

    MOV    A, [ Reg_Y1]
    ADD    A, [ WD_Counter]

    CALL   HCIA_Processing

    ADD    A, [ Reg_Y1]
    MOV    [ Reg_Y1], A

    JC    Loop_Y2C

Loop_Y2:
    CALL   HCIA_Processing

    ADD    A, [ Reg_Y2]
    MOV    [ Reg_Y2], A

    JC    Loop_Y3C

Loop_Y3:
    CALL   HCIA_Processing

    ADD    A, [ Reg_Y3]
    MOV    [ Reg_Y3], A

    JC    Loop_Y4C

Loop_Y4:
    CALL   HCIA_Processing

    ADD    A, [ Reg_Y4]
    MOV    [ Reg_Y4], A

    JC    Loop_Y5C

Loop_Y5:
    CALL   HCIA_Processing

    ADD    A, [ Reg_Y5]
    MOV    [ Reg_Y5], A

    JC    Loop_Y6C

```

```
Loop_Y6:
    CALL    HCIA_Processing

    ADD     A, [ Reg_Y6]
    MOV     [ Reg_Y6], A

    JC      Loop_Y7C
```

```
Loop_Y7:
    CALL    HCIA_Processing
    JC      Skip_NC

    ADD     A, [ Reg_Y7]
    MOV     [ Reg_Y7], A

    JC      Loop_Y8C
```

```
Loop_Y8:
    CALL    HCIA_Processing

    ADD     A, [ Reg_Y8]
    MOV     [ Reg_Y8], A

    JC      Loop_Y1C
    JMP     Loop_Y1
```

```
Skip_NC:
    ADD     A, [ Reg_Y7]
    MOV     [ Reg_Y7], A
    JNC     Loop_Y1
//        JMP     Loop_Y1C
```

```
Loop_Y1C:
```

```
R300_loop_C:
    INC     [ WD_Counter]

    MOV     A, [ Reg_Y1]
    ADD     A, [ WD_Counter]

    CALL    HCIA_Processing

    INC     A
    ADD     A, [ Reg_Y1]
    MOV     [ Reg_Y1], A

    JNC     Loop_Y2
```

```
Loop_Y2C:
    CALL    HCIA_Processing

    INC     A
    ADD     A, [ Reg_Y2]
    MOV     [ Reg_Y2], A

    JNC     Loop_Y3
```

```
Loop_Y3C:
    CALL    HCIA_Processing

    INC     A
    ADD     A, [ Reg_Y3]
    MOV     [ Reg_Y3], A

    JNC     Loop_Y4
```

```
Loop_Y4C:
    CALL    HCIA_Processing

    INC     A
    ADD     A, [ Reg_Y4]
    MOV     [ Reg_Y4], A

    JNC     Loop_Y5

Loop_Y5C:
    CALL    HCIA_Processing

    INC     A
    ADD     A, [ Reg_Y5]
    MOV     [ Reg_Y5], A

    JNC     Loop_Y6

Loop_Y6C:
    CALL    HCIA_Processing

    INC     A
    ADD     A, [ Reg_Y6]
    MOV     [ Reg_Y6], A

    JNC     Loop_Y7

Loop_Y7C:
    CALL    HCIA_Processing
    JC      Skip_C

    INC     A
    ADD     A, [ Reg_Y7]
    MOV     [ Reg_Y7], A

    JNC     Loop_Y8

Loop_Y8C:
    CALL    HCIA_Processing

    INC     A
    ADD     A, [ Reg_Y8]
    MOV     [ Reg_Y8], A

    JNC     Loop_Y1
    JMP     Loop_Y1C
```

```
export    HCIA_Processing
export    HCIA_Setup
```

```
HCIA_Setup:
```

```
        MOV    X, ( Stoppcode-Reg_In-1)
```

```
HCIA_Setup_Loop:
```

```
        INC    [ WD_Counter]
        MOV    A, X
        INDEX  Subst_Table_0
        XOR    [ X + Reg_In], A
```

```
        DEC    X
        JNC    HCIA_Setup_Loop
```

```
        MOV    [ Stoppcode], 0x00
        MOV    [ MVI_A],      HCIA_RAM
        MOV    [ MVI_B],      HCIA_RAM
        MOV    [ WD_Counter], 255
        MOV    X, HCIA_RAM_Mask
        RET
```

```
HCIA_Processing:
```

```
        ADD    [ Reg_In], A           // 7
        AND    A, 0x7E                // 4
        JACC   Operation_table       // 7
```

```
Subst_Table_0:
```

```
DB 0xC4           // Index 0
DB 0x55           // Index 1
DB 0x68           // Index 2
DB 0x09           // Index 3
DB 0x0B           // Index 4
DB 0x81           // Index 5
DB 0x63           // Index 6
DB 0xFC           // Index 7
DB 0x5A           // Index 8
DB 0x45           // Index 9
DB 0x97           // Index 10
DB 0xBD           // Index 11
DB 0x1A           // Index 12
DB 0x27           // Index 13
DB 0x14           // Index 14
DB 0x33           // Index 15
DB 0x8B           // Index 16
DB 0x8E           // Index 17
DB 0x9D           // Index 18
DB 0x23           // Index 19
DB 0x02           // Index 20
DB 0xC1           // Index 21
DB 0x36           // Index 22
DB 0x71           // Index 23
DB 0x24           // Index 24
DB 0x4C           // Index 25
DB 0xCB           // Index 26
DB 0x0F           // Index 27
DB 0xFA           // Index 28
DB 0x83           // Index 29
DB 0x18           // Index 30
DB 0x54           // Index 31
DB 0x01           // Index 32
DB 0xBB           // Index 33
DB 0xE7           // Index 34
DB 0xDE           // Index 35
DB 0x4E           // Index 36
DB 0x4B           // Index 37
DB 0xB4           // Index 38
DB 0x0A           // Index 39
DB 0x38           // Index 40
DB 0xD3           // Index 41
DB 0xD7           // Index 42
DB 0x9E           // Index 43
DB 0x62           // Index 44
DB 0x66           // Index 45
DB 0x4A           // Index 46
DB 0x85           // Index 47
```

DB	0xAD	//	Index	48
DB	0xDC	//	Index	49
DB	0xB1	//	Index	50
DB	0x47	//	Index	51
DB	0x08	//	Index	52
DB	0x03	//	Index	53
DB	0x1D	//	Index	54
DB	0xC8	//	Index	55
DB	0xEC	//	Index	56
DB	0x78	//	Index	57
DB	0xD0	//	Index	58
DB	0xCE	//	Index	59
DB	0xA5	//	Index	60
DB	0x20	//	Index	61
DB	0x60	//	Index	62
DB	0x13	//	Index	63
DB	0x1F	//	Index	64
DB	0x69	//	Index	65
DB	0xF1	//	Index	66
DB	0xFE	//	Index	67
DB	0x59	//	Index	68
DB	0x91	//	Index	69
DB	0xEB	//	Index	70
DB	0x84	//	Index	71
DB	0x76	//	Index	72
DB	0xC7	//	Index	73
DB	0xE3	//	Index	74
DB	0xCF	//	Index	75
DB	0x05	//	Index	76
DB	0x3B	//	Index	77
DB	0xEE	//	Index	78
DB	0x7F	//	Index	79
DB	0x8F	//	Index	80
DB	0xA4	//	Index	81
DB	0x8C	//	Index	82
DB	0xB9	//	Index	83
DB	0xF8	//	Index	84
DB	0x82	//	Index	85
DB	0xB5	//	Index	86
DB	0x12	//	Index	87
DB	0x2B	//	Index	88
DB	0x19	//	Index	89
DB	0x6F	//	Index	90
DB	0x2E	//	Index	91
DB	0x5E	//	Index	92
DB	0x7B	//	Index	93
DB	0xE4	//	Index	94
DB	0x4F	//	Index	95
DB	0xA9	//	Index	96
DB	0x40	//	Index	97
DB	0x80	//	Index	98
DB	0xAE	//	Index	99
DB	0x8D	//	Index	100
DB	0x88	//	Index	101
DB	0xD4	//	Index	102
DB	0xA8	//	Index	103
DB	0xE0	//	Index	104
DB	0x1C	//	Index	105
DB	0x95	//	Index	106
DB	0x3E	//	Index	107
DB	0x92	//	Index	108
DB	0x39	//	Index	109
DB	0x64	//	Index	110
DB	0x00	//	Index	111
DB	0xF0	//	Index	112
DB	0x31	//	Index	113
DB	0x79	//	Index	114
DB	0xFF	//	Index	115
DB	0xF9	//	Index	116
DB	0xD9	//	Index	117
DB	0xDD	//	Index	118
DB	0xD5	//	Index	119
DB	0xFB	//	Index	120
DB	0x77	//	Index	121
DB	0x93	//	Index	122
DB	0x67	//	Index	123
DB	0xF4	//	Index	124
DB	0x26	//	Index	125
DB	0x65	//	Index	126
DB	0x5B	//	Index	127
DB	0x94	//	Index	128
DB	0xFD	//	Index	129
DB	0x5D	//	Index	130
DB	0x1B	//	Index	131
DB	0xCC	//	Index	132
DB	0xE1	//	Index	133
DB	0x56	//	Index	134
DB	0x37	//	Index	135
DB	0xAF	//	Index	136
DB	0x6E	//	Index	137

DB	0xA1	//	Index	138
DB	0x48	//	Index	139
DB	0x44	//	Index	140
DB	0xC9	//	Index	141
DB	0xD8	//	Index	142
DB	0x35	//	Index	143
DB	0x29	//	Index	144
DB	0x75	//	Index	145
DB	0x61	//	Index	146
DB	0x2F	//	Index	147
DB	0xA2	//	Index	148
DB	0x90	//	Index	149
DB	0x7D	//	Index	150
DB	0x6D	//	Index	151
DB	0x11	//	Index	152
DB	0xE2	//	Index	153
DB	0x32	//	Index	154
DB	0x0C	//	Index	155
DB	0xCD	//	Index	156
DB	0x73	//	Index	157
DB	0xBE	//	Index	158
DB	0x25	//	Index	159
DB	0x86	//	Index	160
DB	0xA6	//	Index	161
DB	0x72	//	Index	162
DB	0x51	//	Index	163
DB	0x3D	//	Index	164
DB	0x41	//	Index	165
DB	0x5F	//	Index	166
DB	0xAB	//	Index	167
DB	0xB7	//	Index	168
DB	0x7C	//	Index	169
DB	0xE9	//	Index	170
DB	0x57	//	Index	171
DB	0x50	//	Index	172
DB	0x6C	//	Index	173
DB	0x10	//	Index	174
DB	0xAA	//	Index	175
DB	0x6B	//	Index	176
DB	0xC5	//	Index	177
DB	0x6A	//	Index	178
DB	0xEA	//	Index	179
DB	0xB0	//	Index	180
DB	0xF2	//	Index	181
DB	0xB3	//	Index	182
DB	0x3A	//	Index	183
DB	0x04	//	Index	184
DB	0xE6	//	Index	185
DB	0xC0	//	Index	186
DB	0x34	//	Index	187
DB	0x42	//	Index	188
DB	0xB2	//	Index	189
DB	0x2A	//	Index	190
DB	0x07	//	Index	191
DB	0x28	//	Index	192
DB	0x58	//	Index	193
DB	0x87	//	Index	194
DB	0x3F	//	Index	195
DB	0xF3	//	Index	196
DB	0x74	//	Index	197
DB	0xD6	//	Index	198
DB	0x96	//	Index	199
DB	0x49	//	Index	200
DB	0x7E	//	Index	201
DB	0xDA	//	Index	202
DB	0xC6	//	Index	203
DB	0xA3	//	Index	204
DB	0x43	//	Index	205
DB	0x2D	//	Index	206
DB	0xD2	//	Index	207
DB	0x3C	//	Index	208
DB	0x1E	//	Index	209
DB	0xBF	//	Index	210
DB	0x5C	//	Index	211
DB	0x9A	//	Index	212
DB	0x9F	//	Index	213
DB	0xCA	//	Index	214
DB	0x46	//	Index	215
DB	0x99	//	Index	216
DB	0x53	//	Index	217
DB	0xBC	//	Index	218
DB	0x06	//	Index	219
DB	0xE8	//	Index	220
DB	0x8A	//	Index	221
DB	0xA7	//	Index	222
DB	0xC3	//	Index	223
DB	0xE5	//	Index	224
DB	0xF6	//	Index	225
DB	0x15	//	Index	226
DB	0xD1	//	Index	227

DB	0xEF	//	Index	228
DB	0xA0	//	Index	229
DB	0x52	//	Index	230
DB	0xB6	//	Index	231
DB	0xDB	//	Index	232
DB	0xB8	//	Index	233
DB	0xF5	//	Index	234
DB	0x22	//	Index	235
DB	0x89	//	Index	236
DB	0x0E	//	Index	237
DB	0xBA	//	Index	238
DB	0xF7	//	Index	239
DB	0x70	//	Index	240
DB	0x0D	//	Index	241
DB	0x2C	//	Index	242
DB	0x98	//	Index	243
DB	0xAC	//	Index	244
DB	0x16	//	Index	245
DB	0xDF	//	Index	246
DB	0x30	//	Index	247
DB	0x21	//	Index	248
DB	0x9B	//	Index	249
DB	0x4D	//	Index	250
DB	0x17	//	Index	251
DB	0x7A	//	Index	252
DB	0xC2	//	Index	253
DB	0xED	//	Index	254
DB	0x9C	//	Index	255

Subst_Table_1:

DB	0x14	//	Index	0
DB	0xAE	//	Index	1
DB	0x82	//	Index	2
DB	0xF8	//	Index	3
DB	0xBC	//	Index	4
DB	0x77	//	Index	5
DB	0x19	//	Index	6
DB	0xE8	//	Index	7
DB	0x4E	//	Index	8
DB	0x2D	//	Index	9
DB	0x56	//	Index	10
DB	0xFF	//	Index	11
DB	0xBB	//	Index	12
DB	0xE4	//	Index	13
DB	0x08	//	Index	14
DB	0x5F	//	Index	15
DB	0x9F	//	Index	16
DB	0x76	//	Index	17
DB	0xB7	//	Index	18
DB	0x4D	//	Index	19
DB	0x68	//	Index	20
DB	0x9D	//	Index	21
DB	0x5B	//	Index	22
DB	0xCE	//	Index	23
DB	0x34	//	Index	24
DB	0x27	//	Index	25
DB	0x3D	//	Index	26
DB	0xE7	//	Index	27
DB	0xE1	//	Index	28
DB	0xDC	//	Index	29
DB	0x5D	//	Index	30
DB	0xFA	//	Index	31
DB	0x83	//	Index	32
DB	0x23	//	Index	33
DB	0x0E	//	Index	34
DB	0x51	//	Index	35
DB	0x5E	//	Index	36
DB	0xF6	//	Index	37
DB	0x9B	//	Index	38
DB	0xB2	//	Index	39
DB	0xD0	//	Index	40
DB	0x10	//	Index	41
DB	0x28	//	Index	42
DB	0xB0	//	Index	43
DB	0xEA	//	Index	44
DB	0xC0	//	Index	45
DB	0xD2	//	Index	46
DB	0x50	//	Index	47
DB	0x6E	//	Index	48
DB	0x8D	//	Index	49
DB	0x97	//	Index	50
DB	0x18	//	Index	51
DB	0xD6	//	Index	52
DB	0x47	//	Index	53
DB	0x57	//	Index	54
DB	0x7A	//	Index	55
DB	0x37	//	Index	56

DB	0x13	//	Index	57
DB	0x0B	//	Index	58
DB	0x40	//	Index	59
DB	0x86	//	Index	60
DB	0x38	//	Index	61
DB	0x70	//	Index	62
DB	0xE5	//	Index	63
DB	0xC4	//	Index	64
DB	0x1A	//	Index	65
DB	0x87	//	Index	66
DB	0x11	//	Index	67
DB	0xCB	//	Index	68
DB	0xDE	//	Index	69
DB	0x7E	//	Index	70
DB	0x6D	//	Index	71
DB	0xE3	//	Index	72
DB	0x41	//	Index	73
DB	0x1C	//	Index	74
DB	0xF1	//	Index	75
DB	0x9E	//	Index	76
DB	0x0A	//	Index	77
DB	0x8B	//	Index	78
DB	0x79	//	Index	79
DB	0xD8	//	Index	80
DB	0x22	//	Index	81
DB	0xF3	//	Index	82
DB	0x07	//	Index	83
DB	0x63	//	Index	84
DB	0xD1	//	Index	85
DB	0xCC	//	Index	86
DB	0x45	//	Index	87
DB	0x54	//	Index	88
DB	0x16	//	Index	89
DB	0xFD	//	Index	90
DB	0xA5	//	Index	91
DB	0x92	//	Index	92
DB	0x20	//	Index	93
DB	0xB4	//	Index	94
DB	0xDF	//	Index	95
DB	0x7D	//	Index	96
DB	0x17	//	Index	97
DB	0x61	//	Index	98
DB	0xC7	//	Index	99
DB	0x6F	//	Index	100
DB	0x1B	//	Index	101
DB	0xD4	//	Index	102
DB	0xAF	//	Index	103
DB	0x2B	//	Index	104
DB	0xBA	//	Index	105
DB	0x88	//	Index	106
DB	0x73	//	Index	107
DB	0x43	//	Index	108
DB	0x67	//	Index	109
DB	0x52	//	Index	110
DB	0xAA	//	Index	111
DB	0x0F	//	Index	112
DB	0xF4	//	Index	113
DB	0xD5	//	Index	114
DB	0xAD	//	Index	115
DB	0x53	//	Index	116
DB	0x69	//	Index	117
DB	0x78	//	Index	118
DB	0x30	//	Index	119
DB	0xFC	//	Index	120
DB	0xAB	//	Index	121
DB	0xE6	//	Index	122
DB	0x60	//	Index	123
DB	0x6C	//	Index	124
DB	0x4F	//	Index	125
DB	0x3A	//	Index	126
DB	0x2F	//	Index	127
DB	0xC5	//	Index	128
DB	0x94	//	Index	129
DB	0xA4	//	Index	130
DB	0x0D	//	Index	131
DB	0xDB	//	Index	132
DB	0xA7	//	Index	133
DB	0xB6	//	Index	134
DB	0x58	//	Index	135
DB	0x1D	//	Index	136
DB	0x2A	//	Index	137
DB	0x75	//	Index	138
DB	0xD3	//	Index	139
DB	0x39	//	Index	140
DB	0xE9	//	Index	141
DB	0x8C	//	Index	142
DB	0x29	//	Index	143
DB	0x98	//	Index	144
DB	0xB8	//	Index	145
DB	0xA2	//	Index	146

DB	0x81	//	Index	147
DB	0x7C	//	Index	148
DB	0xF0	//	Index	149
DB	0xF9	//	Index	150
DB	0xC6	//	Index	151
DB	0xBD	//	Index	152
DB	0x72	//	Index	153
DB	0xA9	//	Index	154
DB	0xEB	//	Index	155
DB	0xB3	//	Index	156
DB	0xA8	//	Index	157
DB	0x93	//	Index	158
DB	0xE0	//	Index	159
DB	0x99	//	Index	160
DB	0xD9	//	Index	161
DB	0x85	//	Index	162
DB	0xEC	//	Index	163
DB	0x03	//	Index	164
DB	0xB9	//	Index	165
DB	0xA3	//	Index	166
DB	0xC9	//	Index	167
DB	0x35	//	Index	168
DB	0x5C	//	Index	169
DB	0xF2	//	Index	170
DB	0x42	//	Index	171
DB	0x95	//	Index	172
DB	0x09	//	Index	173
DB	0x26	//	Index	174
DB	0xEE	//	Index	175
DB	0x80	//	Index	176
DB	0xB5	//	Index	177
DB	0x62	//	Index	178
DB	0x33	//	Index	179
DB	0x9C	//	Index	180
DB	0x91	//	Index	181
DB	0x46	//	Index	182
DB	0x06	//	Index	183
DB	0x21	//	Index	184
DB	0x65	//	Index	185
DB	0x04	//	Index	186
DB	0x02	//	Index	187
DB	0x2E	//	Index	188
DB	0x2C	//	Index	189
DB	0x90	//	Index	190
DB	0xC1	//	Index	191
DB	0x48	//	Index	192
DB	0x32	//	Index	193
DB	0x0C	//	Index	194
DB	0x3F	//	Index	195
DB	0x4B	//	Index	196
DB	0x3C	//	Index	197
DB	0xAC	//	Index	198
DB	0xC3	//	Index	199
DB	0x59	//	Index	200
DB	0xDD	//	Index	201
DB	0xDA	//	Index	202
DB	0x7B	//	Index	203
DB	0xCA	//	Index	204
DB	0xCD	//	Index	205
DB	0x9A	//	Index	206
DB	0x6B	//	Index	207
DB	0xED	//	Index	208
DB	0xBF	//	Index	209
DB	0x44	//	Index	210
DB	0x00	//	Index	211
DB	0x55	//	Index	212
DB	0x1E	//	Index	213
DB	0xC8	//	Index	214
DB	0xCF	//	Index	215
DB	0xA0	//	Index	216
DB	0x49	//	Index	217
DB	0x64	//	Index	218
DB	0xBE	//	Index	219
DB	0x84	//	Index	220
DB	0x3E	//	Index	221
DB	0x8E	//	Index	222
DB	0x6A	//	Index	223
DB	0x8F	//	Index	224
DB	0x96	//	Index	225
DB	0xD7	//	Index	226
DB	0x1F	//	Index	227
DB	0xC2	//	Index	228
DB	0x89	//	Index	229
DB	0x7F	//	Index	230
DB	0x74	//	Index	231
DB	0x8A	//	Index	232
DB	0xB1	//	Index	233
DB	0xA1	//	Index	234
DB	0x4A	//	Index	235
DB	0xEF	//	Index	236

DB	0x31	//	Index	237
DB	0xF5	//	Index	238
DB	0x71	//	Index	239
DB	0x05	//	Index	240
DB	0x5A	//	Index	241
DB	0xFB	//	Index	242
DB	0x4C	//	Index	243
DB	0xE2	//	Index	244
DB	0x01	//	Index	245
DB	0x15	//	Index	246
DB	0xF7	//	Index	247
DB	0xA6	//	Index	248
DB	0x66	//	Index	249
DB	0x25	//	Index	250
DB	0x24	//	Index	251
DB	0x12	//	Index	252
DB	0x3B	//	Index	253
DB	0x36	//	Index	254
DB	0xFE	//	Index	255

Operation_table:

```

JMP Operation_00
JMP Operation_01
JMP Operation_02
JMP Operation_03
JMP Operation_04
JMP Operation_05
JMP Operation_06
JMP Operation_07
JMP Operation_08
JMP Operation_09

JMP Operation_10
JMP Operation_11
JMP Operation_12
JMP Operation_13
JMP Operation_14
JMP Operation_15
JMP Operation_16
JMP Operation_17
JMP Operation_18
JMP Operation_19

JMP Operation_20
JMP Operation_21
JMP Operation_22
JMP Operation_23
JMP Operation_24
JMP Operation_25
JMP Operation_26
JMP Operation_27
JMP Operation_28
JMP Operation_29

JMP Operation_30
JMP Operation_31
JMP Operation_32
JMP Operation_33
JMP Operation_34
JMP Operation_35
JMP Operation_36
JMP Operation_37
JMP Operation_38
JMP Operation_39

JMP Operation_40
JMP Operation_41
JMP Operation_42
JMP Operation_43
JMP Operation_44
JMP Operation_45
JMP Operation_46
JMP Operation_47

```

```

JMP Operation_48
JMP Operation_49

JMP Operation_50
JMP Operation_51
JMP Operation_52
JMP Operation_53
JMP Operation_54
JMP Operation_55
JMP Operation_56
JMP Operation_57
JMP Operation_58
JMP Operation_59

JMP Operation_60
JMP Operation_61
JMP Operation_62
JMP Operation_63

```

// Operations read from memory and update registers

```

Operation_00:
// =====
// Op 00 Clocks: 56
MOV  A, [ X_Low ]           // 6
ADD  A, [ Reg_2]           // 6
INDEX Subst_Table_0        // 13
RLC  A                     // 4
ADC  A, [ Reg_3]           // 6
MOV  [ Reg_2], A           // 5
XOR  [ X_High], A         // 8
RET                               // 8

```

```

Operation_01:
// =====
// Op 01 Clocks: 43
MOV  A, [ X_High]          // 6
ADD  A, [ Reg_3]           // 6
RRC  A                     // 4
MOV  [ Reg_3], A           // 5
ADC  A, [ Reg_2]           // 6
ADC  [ X_Low ], A          // 8
RET                               // 8

```

```

Operation_02:
// =====
// Op 02 Clocks: 43
MOV  A, [ X_Low ]           // 6
ADD  A, [ Reg_0]           // 6
RRC  A                     // 4
MOV  [ Reg_0], A           // 5
ADC  A, [ Reg_3]           // 6
ADC  [ X_High], A         // 8
RET                               // 8

```

```

Operation_03:
// =====
// Op 03 Clocks: 48
MOV  A, [ X_Low ]           // 6
XOR  [ Reg_0], A           // 7
XOR  [ Reg_1], A           // 7

ADD  A, [ X_High]         // 7
XOR  [ Reg_2], A           // 7

XOR  A, [ Reg_In]         // 6
RET  // 8

```

```

Operation_04:
// =====
// Op 04 Clocks: 48
MOV  A, [ X_High]         // 6
XOR  [ Reg_0], A           // 7
ADD  [ Reg_1], A           // 7

ADC  A, [ X_Low ]         // 7
XOR  [ Reg_3], A           // 7

ADC  A, [ Reg_In]         // 6
RET  // 8

```

```

Operation_05:
// =====
// Op 05 Clocks: 53
MOV  A, [ X_High]         // 6
XOR  A, [ Reg_In]         // 6
XOR  [ Reg_0], A           // 7
AND  A, HCIA_RAM_Mask     // 4
SWAP A, X                 // 5
ADD  A, 137               // 4
XOR  [ Reg_2], A           // 7
ADD  A, [ Reg_0]          // 6
RET  // 8

```

// Operations read from registers and update memory

```

Operation_06:
// =====
// Op 06 Clocks: 54
MOV  A, [ Reg_0]          // 5
XOR  A, [ Reg_1]          // 6
ADD  [ X_High], A         // 8
INDEX Subst_Table_1      // 13
ADC  A, [ Reg_In]         // 6
ADD  [ X_Low ], A         // 8
RET  // 8

```

```

Operation_07:
// =====
// Op 07 Clocks: 47
MOV  A, [ Reg_3]          // 5
XOR  A, [ Reg_1]          // 6
INDEX Subst_Table_0      // 13
XOR  [ Reg_0], A           // 7
XOR  [ X_Low ], A         // 8
RET  // 8

```

```

Operation_08:
// =====
// Op 08 Clocks: 51
MOV A, [ Reg_0] // 5
XOR A, [ Reg_1] // 6
INDEX Subst_Table_1 // 13
MOV [ Reg_0], A // 5
ADD A, [ Reg_In] // 6
XOR [ X_Low ], A // 8
RET // 8

Operation_09:
// =====
// Op 09 Clocks: 133 (49/49/53/383)
MOV A, [ Reg_3] // 5
ADD A, [ Reg_1] // 6
INDEX Subst_Table_0 // 13
MOV [ Reg_3], A // 5
JC Op_09_1 // 5

XOR A, [ X_Low ] // 7
RET // 8

Op_09_1: ADD A, [ Reg_In] // 6
JNC Op_09_2 // 5
RET // 8

Op_09_2: // Put index reg at random pos
AND A, HCIA_RAM_MaxINX // 4
ADD A, HCIA_RAM // 4
MOV [ MVI_A], A // 5

Op_09_3: // Loop: Average 7.8 * 40 = 312 clocks
ADD [ Reg_2], A // 7
MVI A, [ MVI_A] // 10 Memory end: return zero
ADD [ Reg_0], A // 7
ADC [ Reg_1], A // 7
AND A, %01010001 // 4
JNZ Op_09_3 // 5 Zero 1/8 and at RAM end.

MOV A, [ Reg_0] // 5
RET // 8

Operation_10:
// =====
// Op 10 Clocks: 60
MOV A, [ Reg_0] // 5
ADD [ HCIA_RAM + 105], A // 7
XOR [ X_High], A // 8
MOV A, [ Reg_1] // 5
ADC [ X_Low ], A // 8
MOV A, [ Reg_In] // 6
ADD [ Reg_2], A // 7
XOR A, [ HCIA_RAM + 105] // 6
RET // 8

Operation_11:
// =====
// Op 11 Clocks: 53
MOV A, [ Reg_In] // 5
XOR [ Reg_3], A // 7
ADD A, [ Reg_0] // 6
INDEX Subst_Table_1 // 13
ADD [ Reg_2], A // 7
XOR A, [ X_Low ] // 7
RET // 8

```

// Operations that read memory and registers and compute results

Operation_12:

```
// =====  
// Op 12 Clocks: 49  
MOV A, [ Reg_In] // 5  
OR A, [ Reg_3] // 6  
AND A, [ Reg_0] // 6  
ADD A, [ Reg_2] // 6  
INDEX Subst_Table_1 // 13  
MOV [ Reg_2], A // 5  
RET // 8
```

Operation_13:

```
// =====  
// Op 13 Clocks: 50  
MOV A, [ Reg_In] // 5  
AND A, [ Reg_1] // 6  
ADD A, [ Reg_0] // 6  
INDEX Subst_Table_1 // 13  
MOV [ Reg_0], A // 5  
XOR A, [ X_Low ] // 7  
RET // 8
```

Operation_14:

```
// =====  
// Op 14 Clocks: 55  
MOV A, [ Reg_2] // 5  
AND A, HCIA_RAM_Mask // 4  
SWAP A, X // 5  
OR A, [ Reg_3] // 6  
INDEX Subst_Table_0 // 13  
XOR A, [ X_Low ] // 7  
ADD [ Reg_2], A // 7  
RET // 8
```

Operation_15:

```
// =====  
// Op 15 Clocks:  $63/64*48 + (1/64) * 649 = 57.39$   
MOV A, [ X_High] // 6  
AND A, [ Reg_2] // 6  
ADD A, [ Reg_3] // 6  
DEC X // 4  
JC Op_15_1 // 5  
  
ADD [ Reg_0], A // 7  
ADC A, [ Reg_In] // 6  
RET // 8
```

Op_15_1: // Put index reg at random pos

```
MOV X, 10 // 4  
AND A, HCIA_RAM_MaxINX // 4  
ADD A, HCIA_RAM // 4  
MOV [ MVI_A], A // 5  
MOV [ MVI_B], A // 5
```

Op_15_2: // Loop: Average 10 * 59 = 590 clocks

```
MVI A, [ MVI_A] // 10 Memory end: return zero  
JZ Op_15_3 // 5  
  
ADD A, [ Reg_0] // 6  
XOR A, [ Reg_In] // 6  
INDEX Subst_Table_0 // 13  
MVI [ MVI_B], A // 10  
DEC X // 4  
JNZ Op_15_2 // 5
```

Op_15_3: XOR A, [Reg_1] // 6

```

        AND    A, HCIA_RAM_Mask           // 4
        MOV    X, A                       // 4
        MOV    A, [ X_Low ]               // 6
        ADD    A, [ Reg_In]               // 6
        RET                                // 8

// Exchange operations
Operation_16:
// =====
// Op 16 Clocks: 48
MOV    A, [ Reg_In]                      // 5
SWAP   A, [ Reg_3]                       // 7
SWAP   A, [ Reg_0]                       // 7
SWAP   A, [ Reg_1]                       // 7
ADD    [ X_High], A                      // 8
XOR    A, [ Reg_3]                       // 6
RET                                // 8

Operation_17:
// =====
// Op 17 Clocks: 42
MOV    A, [ Reg_In]                      // 5
SWAP   A, [ Reg_1]                       // 7
SWAP   A, [ Reg_2]                       // 7
XOR    [ X_High], A                      // 8
XOR    A, [ X_Low ]                      // 7
RET                                // 8

Operation_18:
// =====
// Op 18 Clocks: 48
MOV    A, [ Reg_0]                      // 5
XOR    A, [ X_Low ]                      // 7
SWAP   A, [ Reg_In]                     // 7
SWAP   A, [ Reg_2]                     // 7
XOR    [ X_High], A                     // 8
ADD    A, [ Reg_2]                      // 6
RET                                // 8

Operation_19:
// =====
// Op 19 Clocks: 39
MOV    A, [ Reg_2]                      // 5
ADD    A, 235                            // 4
AND    A, HCIA_RAM_Mask                 // 4
SWAP   A, X                              // 5
XOR    A, [ X_Low ]                     // 7
ADD    A, [ Reg_2]                      // 6
RET                                // 8

Operation_20:
// =====
// Op 20 Clocks: 32
MOV    A, [ Reg_2]                      // 5
XOR    A, [ X_Low ]                     // 7
ADD    A, [ Reg_0]                      // 6
ADC    A, [ Reg_In]                     // 6
RET                                // 8

Operation_21:
// =====
// Op 21 Clocks: 32
MOV    A, [ Reg_0]                      // 5
XOR    A, [ X_High]                     // 7
ADD    A, [ Reg_1]                      // 6
XOR    A, [ Reg_In]                     // 6
RET                                // 8

```

```

Operation_22:
// =====
// Op 22 Clocks: 63/64*34 + (1/64)*62 = 34.44
DEC X // 4
JC Op_22_1 // 5
MOV A, [ Reg_3] // 5
ADD A, [ Reg_1] // 6
XOR A, [ Reg_In] // 6
RET // 8

Op_22_1: MOV A, [ Reg_2] // 5
XOR A, [ X_High] // 7
AND A, HCIA_RAM_Mask // 4
SWAP A, X // 5
MOV A, [ Reg_0] // 5
XOR A, [ X_High] // 7
ADD A, [ Reg_1] // 6
ADC A, [ Reg_In] // 6
RET // 8

Operation_23:
// =====
// Op 23 Clocks: 63/64*35 + (1/64)*62 = 35.42
DEC X // 4
JC Op_23_1 // 5
MOV A, [ Reg_0] // 5
ADD A, [ Reg_2] // 6
XOR A, [ X_High] // 7
RET // 8

Op_23_1: MOV A, [ Reg_3] // 5
XOR A, [ X_Low ] // 7
AND A, HCIA_RAM_Mask // 4
SWAP A, X // 5
MOV A, [ Reg_2] // 5
XOR A, [ X_High] // 7
ADD A, [ Reg_3] // 6
ADC A, [ Reg_In] // 6
RET // 8

Operation_24:
// =====
// Op 24 Clocks: (1/2)*29 + (1/4)*52 + (1/4)*79 = 47.25
MOV A, [ X_Low ] // 6
ADD A, [ Reg_1] // 6
RRC A // 4
JC Op_24_1 // 5

ADD [ Reg_0], A // 7
RRC A // 4
ADC A, [ X_High] // 7
JC Op_24_1 // 5

ADD [ Reg_2], A // 7
RRC A // 4
RRC A // 4
XOR A, [ Reg_3] // 6
ADD A, [ Reg_In] // 6

Op_24_1: RET // 8

```

```

Operation_25:
// =====
// Op 25 Clocks: (1/2)*31 + (1/4)*58 + (1/4)*89 = 52.25
MOV A, [ X_High] // 6
ADD A, [ Reg_0] // 6
JC Op_25_1 // 5

ADD [ Reg_1], A // 7
RRC A // 4
RRC A // 4
ADC A, [ X_Low] // 7
JC Op_25_1 // 5

XOR A, [ Reg_3] // 6
ADD [ Reg_2], A // 7
RLC A // 4
RLC A // 4
XOR A, [ Reg_3] // 6
RLC A // 4

Op_25_1: XOR A, [ Reg_In] // 6
RET // 8

```

// Nonlinear register update

```

Operation_26:
// =====
// Op 26 Clocks: 50
MOV A, [ X_High] // 6
OR A, [ Reg_0] // 6
ADD A, [ Reg_1] // 6
INDEX Subst_Table_0 // 13
XOR A, [ Reg_In] // 6
MOV [ Reg_0], A // 5
RET // 8

```

```

Operation_27:
// =====
// Op 27 Clocks: 44
MOV A, [ X_Low ] // 6
XOR A, [ Reg_2] // 6
ADD A, [ Reg_3] // 6
INDEX Subst_Table_0 // 13
MOV [ Reg_3], A // 5
RET // 8

```

```

Operation_28:
// =====
// Op 28 Clocks: 34
MOV A, [ Reg_0] // 5
AND A, [ Reg_3] // 6
ADD A, [ Reg_1] // 6
XOR A, 0xAA // 4
MOV [ Reg_1], A // 5
RET // 8

```

```

Operation_29:
// =====
// Op 29 Clocks: 45
MOV A, [ Reg_0] // 5
AND A, HCIA_RAM_Mask // 4
SWAP A, X // 5
OR A, [ Reg_1] // 6
ADD A, [ Reg_2] // 6
MOV [ Reg_2], A // 5
ADC A, [ Reg_In] // 6
RET // 8

Operation_30:
// =====
// Op 30 Clocks: 45
MOV A, [ Reg_2] // 5
AND A, HCIA_RAM_Mask // 4
SWAP A, X // 5
OR A, [ Reg_0] // 6
ADD A, [ Reg_3] // 6
MOV [ Reg_3], A // 5
ADC A, [ Reg_In] // 6
RET // 8

Operation_31:
// =====
// Op 31 Clocks: 46
MOV A, [ Reg_1] // 5
AND A, HCIA_RAM_Mask // 4
SWAP A, X // 5
OR A, [ Reg_3] // 6
ADD A, [ Reg_In] // 6
MOV [ Reg_In], A // 5
ADC A, [ X_Low ] // 7
RET // 8

Operation_32:
// =====
// Op 32 Clocks: 46
MOV A, [ Reg_2] // 5
AND A, HCIA_RAM_Mask // 4
SWAP A, X // 5
OR A, [ Reg_0] // 6
ADD A, [ Reg_In] // 6
MOV [ Reg_In], A // 5
ADC A, [ X_Low ] // 7
RET // 8

Operation_33:
// =====
// Op 33 Clocks: (1/2)*31 + (1/4)*53 + (1/4)*110 = 56.25
MOV A, [ X_High] // 6
ADD A, [ Reg_1] // 6
JC Op_33_1 // 5

ADD [ Reg_0], A // 7
RRC A // 4
ADC A, [ Reg_3] // 6
JC Op_33_1 // 5

XOR A, [ Reg_2] // 6
INDEX Subst_Table_1 // 13
ADD [ Reg_0], A // 7
RLC A // 4
RLC A // 4
ADD [ X_Low ], A // 8

```

```

        AND    A, HCIA_RAM_Mask           // 4
        SWAP  A, X                       // 5
        OR    A, [ Reg_0]                // 6

Op_33_1:  XOR    A, [ Reg_In]            // 6
        RET                                // 8

Operation_34:
        // =====
        // Op 34 Clocks: 255/256*44 + 1/256*2280 = 52.73
        MOV   A, [ Reg_3]                // 5
        AND   A, [ Reg_In]              // 6
        SUB   A, [ Reg_1]                // 6
        JZ    Op_34_1                    // 5   JMP (1/256)

        ADD   [ X_High], A                // 8
        MOV   A, [ X_High]                // 6
        RET                                // 8

Op_34_1:  // Put index reg at random pos
        // A is zero
        MOV   X, 35                       // 4
        MOV   A, [ Reg_0]                 // 5
        AND   A, HCIA_RAM_MaxINX         // 4
        ADD   A, HCIA_RAM                 // 4
        MOV   [ MVI_A], A                 // 5
        MOV   [ MVI_B], A                 // 5

Op_34_2:  // Loop: Average 35 * 65 = 2275 clocks
        MVI   A, [ MVI_A]                 // 10   Memory end: return zero
        JZ    Op_34_3                     // 5

        ADD   [ Reg_3], A                 // 7
        MOV   A, [ Reg_3]                 // 5
        XOR   A, [ Reg_In]                // 6
        INDEX Subst_Table_1               // 13
        MVI   [ MVI_B], A                 // 10
        DEC   X                            // 4
        JNZ   Op_34_2                     // 5

Op_34_3:  XOR   A, [ Reg_1]               // 6
        ADD   A, [ Reg_3]                 // 6
        AND   A, HCIA_RAM_Mask            // 4
        MOV   X, A                         // 4
        MOV   A, [ X_Low ]                 // 6
        ADD   A, [ Reg_In]                 // 5
        RET                                // 8

Operation_35:
        // =====
        // Op 35 Clocks: 26
        MOV   A, [ Reg_In]                // 5
        ADD   A, [ Reg_0]                 // 6
        XOR   [ Reg_1], A                  // 7
        RET                                // 8

Operation_36:
        // =====
        // Op 36 Clocks: 26
        MOV   A, [ Reg_In]                // 5
        ADD   A, [ Reg_1]                 // 6
        XOR   [ Reg_3], A                  // 7
        RET                                // 8

```

```

Operation_37:
// =====
// Op 37 Clocks: 26
MOV A, [ Reg_In] // 5
SUB A, [ Reg_3] // 6
XOR [ Reg_0], A // 7
RET // 8

```

```

Operation_38:
// =====
// Op 38 Clocks: 27
MOV A, [ Reg_In] // 5
XOR A, [ X_Low] // 7
ADD [ Reg_2], A // 7
RET // 8

```

```

Operation_39:
// =====
// Op 39 Clocks: 27
MOV A, [ Reg_In] // 5
XOR A, [ X_High] // 7
SUB [ Reg_3], A // 7
RET // 8

```

```

Operation_40:
// =====
// Op 40 Clocks: 40
MOV A, [ Reg_In] // 5
ADD A, [ X_High] // 7
INDEX Subst_Table_1 // 13
XOR [ Reg_0], A // 7
RET // 8

```

```

Operation_41:
// =====
// Op 41 Clocks: 40
MOV A, [ Reg_In] // 5
SUB A, [ X_Low ] // 7
INDEX Subst_Table_0 // 13
XOR [ Reg_1], A // 7
RET // 8

```

```

Operation_42:
// =====
// Op 42 Clocks: 38
MOV A, [ Reg_3] // 5
AND A, [ X_Low ] // 7
ADD A, [ Reg_In] // 6
XOR [ Reg_2], A // 7
MOV A, [ Reg_2] // 5
RET // 8

```

```

Operation_43:
// =====
// Op 43 Clocks: 37
MOV A, [ Reg_0] // 5
OR A, [ Reg_1] // 6
ADD A, [ Reg_In] // 6
XOR [ Reg_3], A // 7
MOV A, [ Reg_3] // 5
RET // 8

```

```

Operation_44:
// =====
// Op 44 Clocks: 39
MOV A, [ Reg_3] // 5
OR A, [ Reg_2] // 6
ADD A, [ Reg_In] // 6
XOR [ X_Low], A // 8
MOV A, [ X_Low] // 6
RET // 8

Operation_45:
// =====
// Op 45 Clocks: 33
MOV A, [ Reg_1] // 5
ADD A, [ Reg_0] // 6
SBB [ X_Low ], A // 8
MOV A, [ X_Low ] // 6
RET // 8

Operation_46:
// =====
// Op 46 Clocks: 25
MOV A, [ Reg_2] // 5
AND A, [ Reg_1] // 6
ADD A, [ Reg_In] // 6
RET // 8

Operation_47:
// =====
// Op 47 Clocks: 63/64*42 + (1/64)*61 = 42.31
DEC X // 4
JC Op_47_1 // 5
MOV A, [ Reg_2] // 5
OR A, [ X_Low ] // 7
SUB A, [ X_High] // 7
XOR A, [ Reg_In] // 6
RET // 8

Op_47_1: MOV A, [ Reg_0] // 5
XOR A, [ X_High] // 7 // OK, High[-1] =
// = Low[ HCIA_RAM_Mask]

AND A, HCIA_RAM_Mask // 4
MOV X, A // 4
MOV A, [ Reg_2] // 5
XOR A, [ X_High] // 7
SUB A, [ Reg_2] // 6
ADC A, [ Reg_In] // 6
RET // 8

```

```

Operation_48:
// =====
// Op 48 Clocks: 63/64*40 + (1/64)*82 = 40.66
DEC X // 4
JC Op_48_1 // 5
MOV A, X // 4
XOR A, [ Reg_0] // 6
XOR A, [ X_Low ] // 7
ADD A, [ Reg_In] // 6
RET // 8

Op_48_1: MOV A, [ Reg_2] // 5
ADD A, [ X_High] // 7 // OK, High[-1] =
// = Low[ HCIA_RAM_Mask]

AND A, HCIA_RAM_Mask // 4
MOV X, A // 4
MOV A, [ Reg_1] // 5
ADD A, [ X_High] // 7
XOR A, [ X_Low ] // 7
INDEX Subst_Table_0 // 13
SUB [ Reg_2], A // 7
ADD A, [ Reg_In] // 6
RET // 8

Operation_49:
// =====
// Op 49 Clocks: 53
MOV A, [ Reg_In] // 5
AND A, HCIA_RAM_Mask // 4
MOV X, A // 4
MOV A, [ X_Low] // 6
ADD [ Reg_2], A // 7
XOR A, [ Reg_0] // 6
INDEX Subst_Table_0 // 13
RET // 8

Operation_50:
// =====
// Op 50 Clocks: 53
MOV A, [ HCIA_RAM + 73] // 5
AND A, HCIA_RAM_Mask // 4
MOV X, A // 4
MOV A, [ X_Low] // 6
ADD [ Reg_1], A // 7
XOR A, [ Reg_0] // 6
INDEX Subst_Table_0 // 13
RET // 8

Operation_51:
// =====
// Op 51 Clocks: 44
MOV A, [ HCIA_RAM + 73] // 5
ADD [ Reg_2], A // 7
XOR A, [ Reg_0] // 6
INDEX Subst_Table_0 // 13
MOV [ HCIA_RAM + 73], A // 5
RET // 8

Operation_52:
// =====
// Op 52 Clocks: 44
MOV A, [ HCIA_RAM + 34] // 5
ADD [ Reg_0], A // 7
SUB A, [ Reg_3] // 6
INDEX Subst_Table_1 // 13
MOV [ HCIA_RAM + 34], A // 5
RET // 8

```

```

Operation_53:
// =====
// Op 53 Clocks: 33
MOV A, [ HCIA_RAM + 12] // 5
ADD [ Reg_In], A // 7
SUB A, [ Reg_3] // 6
XOR A, [ X_Low ] // 7
RET // 8

Operation_54:
// =====
// Op 54 Clocks: 107 (36/36/53/303)
MOV A, [ Reg_0] // 5
ADD A, [ Reg_2] // 6
MOV [ Reg_0], A // 5
JC Op_54_1 // 5

XOR A, [ X_High] // 7
RET // 8

Op_54_1: ADC A, [ Reg_In] // 6
INDEX Subst_Table_1 // 13
JNC Op_54_2 // 5
RET // 8

Op_54_2: // Put index reg at random pos
AND A, HCIA_RAM_MaxINX // 4
ADD A, HCIA_RAM // 4
MOV [ MVI_A], A // 5
MOV [ MVI_B], A // 5
MVI A, [ MVI_A] // 10
ADD A, [ Reg_2] // 6
ADC [ Reg_3], A // 7
RLC A // 4
ADC A, 186 // 4
MVI [ MVI_B], A // 10
MVI A, [ MVI_A] // 10 Memory end: return zero
JZ Op_54_3 // 5

ADD A, [ Reg_0] // 6
ADC [ Reg_2], A // 7
RLC A // 4
ADC A, 218 // 4
MVI [ MVI_B], A // 10
MVI A, [ MVI_A] // 10 Memory end: return zero
JZ Op_54_3 // 5

ADD A, [ Reg_2] // 6
ADC [ Reg_1], A // 7
RLC A // 4
ADC A, [ X_Low] // 7
MVI [ MVI_B], A // 10
MVI A, [ MVI_A] // 10 Memory end: return zero
JZ Op_54_3 // 5

ADD A, [ Reg_1] // 6
ADC [ Reg_0], A // 7
RLC A // 4
RLC A // 4
ADC A, 158 // 4
MVI [ MVI_B], A // 10
MVI A, [ MVI_A] // 10 Memory end: return zero
JZ Op_54_3 // 5

ADD A, [ Reg_1] // 6
RLC A // 4

```

```

                MVI    [ MVI_B], A                // 10
Op_54_3:      MOV    A, [ Reg_3]                // 5
                ADD    A, [ Reg_In]             // 6
                RET                                // 8

```

```

Operation_55:
// =====
// Op 55 Clocks: 34
MOV    A, [ HCIA_RAM + 7]                // 5
ADD    [ X_Low], A                       // 8
SUB    A, [ Reg_3]                       // 6
XOR    A, [ X_High]                       // 7
RET                                // 8

```

```

Operation_56:
// =====
// Op 56 Clocks: 34
MOV    A, [ HCIA_RAM + 17]               // 5
XOR    A, [ Reg_In]                     // 6
ADD    [ X_Low], A                       // 8
XOR    A, [ X_High]                       // 7
RET                                // 8

```

```

Operation_57:
// =====
// Op 57 Clocks: 39
MOV    A, [ HCIA_RAM + 65]               // 5
XOR    A, [ Reg_2]                       // 6
XOR    A, [ X_High]                       // 7
INDEX Subst_Table_1                      // 13
RET                                // 8

```

```

Operation_58:
// =====
// Op 58 Clocks: 44
MOV    A, [ Reg_0]                       // 5
XOR    A, [ Reg_1]                       // 6
XOR    A, [ Reg_2]                       // 6
XOR    A, [ Reg_3]                       // 6
INDEX Subst_Table_0                      // 13
RET                                // 8

```

```

Operation_59:
// =====
// Op 59 Clocks: 38
MOV    A, X                               // 4
ADD    A, [ X_High]                       // 7
INDEX Subst_Table_0                      // 13
ADD    A, [ Reg_In]                       // 6
RET                                // 8

```

```

Operation_60:
// =====
// Op 59 Clocks: 26
MOV    A, [ Reg_3]                       // 5
ADD    A, [ X_Low ]                       // 7
ADC    A, [ Reg_In]                       // 6
RET                                // 8

```

```

Operation_61:
// =====
// Op 61 Clocks: 73
MOV   A, [ Reg_2]           // 5
Op_61_1:
DEC   X                     // 4   Stop: 1/30
JC    Op_61_2               // 5
ADD   A, [ X_High]         // 7
MOV   [ X_High], A         // 6
JNC   Op_61_1               // 5   Stop: 1/2
XOR   A, [ Reg_In]         // 6
RET                                // 8

Op_61_2:
AND   A, HCIA_RAM_Mask     // 4
MOV   X, A                  // 4
ADD   A, [ Reg_In]         // 6
RET                                // 8

Operation_62:
// =====
// Op 62 Clocks: 40
MOV   A, [ X_Low ]         // 6
ADD   A, [ Reg_0]          // 6
RLC   A                     // 4
RLC   A                     // 4
MOV   [ Reg_0], A         // 5
ADC   A, [ X_High]         // 7
RET                                // 8

Operation_63:
// =====
// Op 63 Clocks: 40
MOV   A, [ X_High]         // 6
ADD   A, [ Reg_1]          // 6
RLC   A                     // 4
RLC   A                     // 4
MOV   [ Reg_1], A         // 5
XOR   A, [ X_Low ]         // 7
RET                                // 8

```

The HCIA Encryption System

In an encryption system the HCIA should be viewed as a part of the solution. The complete solution will also need additional building blocks. In an implementation the HCIA is run as a stream cipher. The input to the HCIA would be the secret cipher key, the plaintext, and the generated ciphertext. Preferably shall the plaintext be padded with randomness from a good TRNG.

The actual encryption may take place using any simple encryption function. The recommendation is to avoid a linear encryption function; the encryption function shall have some resistance against attack. A simple substitution or a four-square substitution is recommended. It is not necessary to use a high-complex substitution like the DES. We note that, even if the selected function will provide some resistance against cryptanalysis, it cannot be the sole basis of security of the system.

The encryption function has access to an internal key source. This is the HCIA subsystem. The HCIA subsystem will for each input plaintext byte produce a number of input key-bytes. The encryption system will encrypt using the information as a key source.

From OTP discussions you might remember that a single perfect-entropy byte can be used with the XOR operator to yield perfect secrecy for a plaintext byte. We will not be that lucky. Due to this we use a more complex operator than the XOR operator, and we output several bytes from the HCIA subsystem, for each plaintext byte that we encrypt. A security factor is used: by obtaining much more information from the HCIA subsystem than an "OTP" (or information theoretic) discussion would suggest, we enforce that a successful cryptanalysis of the entire system must also include successful cryptanalysis of the HCIA subsystem.

The corresponding decryption function might be obscure. Consider the following update algorithm:

- 1) Assume that you have 4 bytes of internal key.
- 2) Use this key to modify or update (change) the invertible four-square mapping, that is used for encryption and decryption.
- 3) Use the invertible four-square to encrypt two bytes of plaintext or decrypt two bytes of ciphertext.
- 4) We now have, independent of if we do encryption or decryption two plaintext bytes and two ciphertext bytes.
- 5) Run the HCIA subsystem with plaintext byte 1,2 and then ciphertext byte 1,2. Perform this function identically in encryption mode and decryption mode.
- 6) We now obtain four new bytes of internal key from the HCIA subsystem.

To further discuss an implementation of a cipher system, we may encrypt in several layers, similar to the DES iterations. If we reverse the order in which the plaintext/ciphertext is input to the encryptor -- so that each odd iteration is performed backwards -- this will bind the information in the block together using the feedback in the HCIA system. A further property is that, for intermediate iterations, both the plaintext and the ciphertext side is protected by other encryptions. This protects access to information in the intermediate iterations by cryptanalysis methods.

Previously we reached the conclusion, that if the cipher would permit us to have its cryptanalytic solutions arbitrary or randomly positioned, in the set of the integer functions, the resulting cipher system would be secure. How should we accomplish this? Can it be done at all?

Why is it that the details of the operations don't matter much for the security of the system?? It is well known, that for all other kind of systems, the details matter a great deal!

Suppose that we change the implementation of an operation. Take operation 14. At the end it add the accumulator to "Reg_2". In this context we could have used any of the four registers. If we change the contents of an operation then will the algorithm, defined by the input, also change. The input software, written in a general purpose language "A" is now written in a general purpose language "B". But from a security point of view we merely need that the input is interpreted as a GENERAL PURPOSE LANGUAGE -- it does not matter much which. This freedom let us adopt, and adjust, the HCIA operations to work well for different applications.

In fact, detailed analysis of all the HCIA operations cannot get us anywhere. The reason is that the set of all operations form a set, enabling full computability. As with all other microcontrollers it is not the microcontroller instruction set, that define the task, that the microcontroller perform. It is the software that define what the MC perform. We conclude that no cryptanalyst can gain information about the encryption by a study of the operation list.

It is the software that define the output, and the capabilities, of the cipher. The software, that we keep secret, is generated by the plaintext and the secret key of the system.

Codebreaking: Questions and Assumptions

Before we begin discussing security and properties of the HCIA cipher, or a HCIA subfunction of a cipher implementation, there are three distinct problems with the HCIA cipher compared to other encryption systems that we commonly find today. We begin by stating these so that The Reader may check if he has comprehended these fundamental facts.

- 1) The first fundamental difference is that the HCIA is a varying cipher, and not a static cipher solution. The strength will come from that an imagined opponent, that we call the cryptanalyst, will find it difficult to follow, comprehend, adopt, and adjust according to the "movements" of the HCIA cipher. Traditionally, cryptanalytic strength comes from a static, invariant function, that is performed identically, independent of what plaintext or what cipherkey is input to the cipher function/algorithm. Due to this different concept of construction, new concept of discussion and theoretical treatment will be advantageous or even necessary. The usual arguments may no longer apply.
- 2) The HCIA cipher do not in itself contain the elements of the functions generated. Rather, these are generated indirectly by an interpreter model. Readers with comprehensive background in computer science may easily follow this line of thought, while it may appear strange to others. Especially when possible or proposed attacks are discussed it is an advantage if the linguistic model, with a discussion of the complexity of the the grammar generated by the language, can be held in parallel with a discussion on the corresponding computational details. We note that, even if not often discussed in the area of encryption technology, this knowledge is most easy to acquire [102].
- 3) Finally, when we discuss the HCIA cipher we do not discuss an instance of a cipher, but a class. From a cryptanalytic point of view this is indeed unusual, and the difficulties seems to be enormous or overwhelming, making the task threatening or looking ridiculous. But by discussing a class, and not an instance, we also gain a different perspective, and the language reach a higher meta-level, where it may have a higher describability, possibly even making our task easier.

When analysing the resistance against cryptanalysis for the HCIA cipher, or rather for a cipher implementation including a subsystem built according to the HCIA technology, the most preferred result would be a mathematical result that codebreaking would be impossible. Before we perform detailed analysis after these lines, we begin with a study of simpler aspects of the cipher.

- -

The most basic entity in a cipher system is the cipher algorithm. It is important to understand the concept of cipher algorithm in a HCIA system. The computation sequence is controlled by the operation selector. This subfunction is driven by the input to the HCIA system. In a cipher application this will consist of the combination of the input plaintext, the output ciphertext, the secret cipher key, and randomness taken from a true random number generator (TRNG). This directly imply that this information will not likely be identical twice. We conclude that the cipher algorithm will be different for different networks (different users, different customers) as they will select different keys. We conclude that each individual transmission or message will be secured using a unique cipher algorithm.

Based upon these observations, there must follow two conclusions:

- A1) It may be true that there might exist a general cryptanalysis method that break all ciphers that use the HCIA technology.
- A2) If (A1) is false then each individual cipher algorithm must be attacked individually. Some effort will be needed by a cryptanalysis group/department/bureau to gain access to the plaintext in each individual transmission. We conclude that it must be true that only a limited number of transmissions may be broken each calendar year, and all other transmissions will remain secure.

- -

Further information about the difficulties in cryptanalysing the HCIA subsystem may be obtained by observing that the actual used cipher algorithm remain secret. For the HCIA we may select the sequence of operations as an encoding of the cipher algorithm used. If the cryptanalyst need this information, it must be deduced by a cryptanalytic attack on available output. Based upon this we conclude that:

- B1) It is true that it is possible to deduce the sequence of HCIA-operations by a cryptanalytic attack.
- B2) After (B1) this may lead to a successful cryptanalytic attack on the complete cipher.
- B3) If not (B1) is it true that it may be possible to perform a successful cryptanalytic attack without obtaining the details about the HCIA instruction sequence.

- -

For the HCIA cipher we have stated that effort should be made that let the computational power of the HCIA implementation be as similar as possible to an ordinary general purpose computer, so that the HCIA subfunction simulates the theoretical Turing machine. On this we may state that:

- C1) It is true that the theoretical Turing machine model have the property that it is not possible to deduce a property of the input by an observation of the output.
- C2) Even if (C1) will be true in a theoretical setting, it cannot be true in the real world, as the Turing machine model target at infinite computation sequences. A finite computation sequence correspond to a limited computation, and the theoretical discussion do not apply.
- C3) Even if (C1) will be true in a theoretical setting, it cannot be true for the HCIA cipher, as this must be implemented using limited resources. Consequently we may see the HCIA as an ordinary algorithm implementation, and the "non-algorithmic" arguing do not apply.
- C4) Even if the HCIA implementation is secure, it may be impossible to construct the invertible cipher function, that actually encrypt the plaintext symbols, without introducing some weakness into the system. Due to this the security of the HCIA don't apply to a real implementation.

- -

Some cryptanalysis methods is based upon a careful study of the statistics of the output ciphertext symbols. Such analyse often consist of building a statistical model of the cipher algorithm, and then obtaining information about the cipher by processing a preferably large number of similar intercepted ciphertexts. Since the HCIA system will generate a new algorithm, for each such message, we conclude:

- D1) Any successful statistical attack must be based upon a property of the implemented cipher solution that remain invariant for each encryption.
- D2) It must be true that no statistical method may work on the HCIA subsystem, as the implemented cipher algorithm is different for each encrypted message, and only one sample can be obtained from each encryption algorithm.

- -

From a linguistic perspective we may say that the HCIA function define the language in which the encryption takes place. We may see this as an interpretation process. The cryptanalyst must now build a corresponding description. On this we may state that:

- E1) Even if it is difficult, the cryptanalyst will deduce a description of the cipher. But this description cannot be complete, so that every property of the cipher, all properties of the input plaintext, etc., is included in this model. We conclude that it must be true that this description must be partial, so that at least some part of the properties of the system will not be available in the cryptanalytic attack.
- E2) The cryptanalyst may work in a language defined by the HCIA system, and find a working solution there, or it must be true that he must work in a higher language, with a higher describability, and find his solution in this context.

- -

From our initial discussion on the function space of cipher functions and code-breaking we may ask ourselves how rich the function space generated by the HCIA cipher is, and where we will expect to find the corresponding cryptanalytic solutions. We will in particular ask ourselves if:

- F1) If it is true that the HCIA subfunction can generate a computation space so dense that it includes a sufficient proportion of all possible computations, that can be performed using any computational sequence of similar length.
- F2) If it is reasonable to expect that, for every problem defined by one particular computational sequence, that there must be another computational sequence that solves this question, or if it in general this is not the case.

- -

In general we may ask how a cipher should be constructed; if an absolutely secure system can, at all, be built. In particular we will ask if:

- G1) The HCIA-method is a general method that yield secure systems,
- G2) or, if not, if we can build even more complex systems, that are even more difficult to cryptanalyse, and possible will these systems be secure, even if the present HCIA is not,
- G3) or, if this can at all be done. Possible there must for every possible computation, that is a cipher, exist another computation that enable a successful break.

- -

The question on if a "brute-force" scan for the secret key consists of a valid (working) cryptanalysis method still remain:

H1) Evidently, searching for the key at random in the keyspace, might work, and consequently no cipher can ever be proven to be secure.

- -

Our last proposition is that, if we for sake of argument assume that we find a theoretical discussion abstract, dangerous, or merely meaningless; can we make a comparison between the difficulty of cryptanalysing the HCIA system, and some real-world-problem, to let us indirectly conclude a sufficient minimum security level for the HCIA system. In particular we ask us:

K1) If we can, using our present knowledge, construct such a comparison.

K2) If we may extrapolate this knowledge into the unknown future.

- -

Codebreaking: Detailed Analysis

Due to that the HCIA subfunction intentionally is constructed so that it mimic the Turing machine, there is a great abundance of theoretical material that may come into play, when we work with the security of the cipher. But our focus must be the quality and the applicability of our models and the resulting conclusions. Therefore we will not aim for the strongest or most elegant result. Instead we will aim for reliability and security. It is much better to have a rather weak result, that we can fully rely on, compared to a security proof that offer perfect security, but that may not be applicable in all situations or under all operational conditions.

We should also note that, previously, no progress has been made in the theoretical assessment of the security of ciphers. Previous there have been only partial results, such that a specific attack method/algorithm cannot be applied. There have clearly been substantial negative material published in the form of weaknesses for in many different ciphers. Some researchers believe that the task of obtaining positive results cannot at all be done. We conclude that any progress -- any what-so-ever -- would be a substantial step in the right direction.

- -

A Micro Model

Suppose that we observe a single memory cell, one byte, somewhere in the HCIA memory. Suppose that an update has just taken place, and the cell is assigned a value of 135. How should we interpretate the number 135? The number is a result of some input to the cipher, plaintext, secret key, randomness from a TRNG. This information is processed by applying HCIA operations. One operation just assigned 135 to the cell. We conclude that the bit pattern of "135" is the result of a function sequence, that have assigned this number. We may this look upon the number in two ways: a piece of information, a number in range $0 \leq n \leq 255$ with a maximum of 8 bits of information; or we may look upon the number as an instance of a function chain. If the function chain where to be repeated with the same input, the same number will result. So we may say that either may the memory cell store a maximum of eight bits of information, or a chain of performed calculations.

After the update operation there follow a sequence, possibly long, where no update takes place at this cell. Evidently the cell contain the result of the previously executed function chain. But the possibility exist that some other key, randomness, or input plaintext would have produced a different update chain, and that the the number 135 would have been rather quickly replaced.

We shall conclude ??? that a cell not only contain information about the functional sequence, that generated that number, but also information about the following update sequence that did not update this particular location, but instead performed updates at other locations!?

A difficult concept is the relative rate, by which these events follow. In particular how much information and how many variants are there, for a particular update sequence, and how many possible update sequences are there to choose this particular update sequence from?

Suppose, for sake of example, that we model a function with a function table. The function have four input bits, and four output bits. Wee see that the function can produce only one number in 16, each time it is applied. If it is applied ten times it will produce 40 bits of output. That will be one binary string in 2^{40} .

A function table can be built for the function with 16 rows, one for each input, and with a four-bit word on each row. We can thus store, or implement, any such function by storing $16 \cdot 4 = 64$ bits. There are thus 2^{64} different such functions. We see that when we input four bits and obtain four bits, anyone of 2^{64} functions might have been applied.

If we proceed with applying the functions ten times, and if we let the functions be independent of each other (ten different functions), then the resulting 40 bit word is produced by selecting one function in a space of $(2^{64})(2^{64})(2^{64})(2^{64})(2^{64})(2^{64})(2^{64})(2^{64})(2^{64})(2^{64}) \dots$ that will be 2^{640} .

An essential problem for a cryptanalyst, when he encounter the HCIA system, is that it generate these kind of possibilities at high speed. In a "normal" cryptanalysis model, the cryptanalyst is supposed to keep both eyes on the encryption function used. But for the HCIA system this functional space, the space from which the functions are selected (not the combination of functions actually selected); the properties of the abilities of the system, compared to the properties of the functional sequence selected; comprise the essential problem that the cryptanalyst need to solve.

Solving this may be an impossible task. The number of 135, above, was stored in a single byte. When this byte is reused, a maximum of 8 bits is inserted into the operation of the system. Possibly will this model of description, 8 bits of information, be shorter and easier to work with, compared to the functional expression that actually generated this particular number?? But this is a dead-end for a cryptanalyst (directly lethal!), as in this case 8 bits of new information just entered the encryption. It will be good for the OTP encryption of one plaintext byte, so if we/cryptanalyst choose to use the information view of the 135 number, then we just lost one intercepted ciphertext byte, that must now be used to deduce the "135".

Instinctively we feel that, for most HCIA implementations (possibly not the particular example implementation above, due to the (very) small memory) the sequence of HCIA-operations-numbers will be a measure of a minimum information of how the update sequence takes place. Under reasonable assumptions this information flow will be less than the flow of information between HCIA-registers and HCIA-memory.

According to the discussion above we may arrange, by a design decision, so that the information flow in this thought-model is much larger than the information flow that any cryptanalyst may obtain by studying the intercepted ciphertext.

As an example, using the suggested encryption implementation in a chapter above, if we iterate the HCIA-implementation 4 times on a block of input, and if we run both the "plaintext" and the "ciphertext" bytes through the HCIA system, and if we assume a 64 operations HCIA system, then the minimum information in the operations will be $2 \cdot 4 \cdot 6$ bits = 48 bits for each 8-bit plaintext and ciphertext byte. Preferably we will arrange so that the HCIA is iterated. Suppose that the probability of iterating (input: old plaintext/ciphertext bytes) is set to a moderate value of 3/4. Then we will on average perform 4 update operations for each input. We now have 4 iterations times two byte input times 4 iterations times six bits to specify the chain of HCIA-operations. That is 192 bits. At most the cryptanalyst may obtain 8 bits of known/guessed plaintext, and 8 bits of intercepted ciphertext. That is only 16 bits. He lacks, desperately, 176 bits. Worse, he lacks 176 bits for each and every plaintext byte, that is processed by the system.

From this point on there will be no difficulty of applying statistical methods, such as actually measure the information flow rates in all part of the HCIA system and of the cipher, and adjusting the system so that a reasonable security margin of 10-100 times the information flow of the ciphertext is achieved.

The discussion of the example above also reveal a potential problem with the HCIA technology: sufficient size, complexity, and space. The solution might take various forms, but in essence, the HCIA must not be run on "small" inputs. A 512 byte string of true randomness, from a good TRNG, make encryption of a message with the HCIA somewhat easier...

The conclusion of the previous discussion shall not be that the HCIA cipher is secure, but merely that we may easy force the cryptanalyst of doing the cryptanalysis the hard way, i.e. detailed analyse of the HCIA-operations, obtaining the actual sequence of operations, and so on.

We may however conclude that the proposition (C4) must be false.

- -

Guess my Software!

We now continue with a discussion of what we may say, generally, about a software that run on a general purpose computer, given some output. This example/proof is by Jesper Jansson, Dept. of Computer Science, Lund University.

Suppose that a cryptanalyst is trying to identify a function $C=F_1(M)$ by using a (finite) table of corresponding pairs of input and output $\{M_i, C_i\}$. For special functions $F()$, for example linear functions, this problem may be solved efficiently. That can not be the case for general functions. As an example set $C=F_1(M)=|M|$ and compare $F_1(M)$ to $F_2(M)$ where $F_2(M)=\{|M| \text{ if } M \neq M_0, \text{ but } "0" \text{ if } M=M_0\}$. It is then clear that no finite number of experiments $\{M_i, C_i\}$ will suffice for the cryptanalyst to separate the two functions $F_1()$ and $F_2()$ from each other, as the arbitrary number M_0 may have been set to any value. (Further, if the M_0 is found, we will not know if our $F()$ really is the $F_2()$, as it could also be a $F_3()$, $F_4()$,...) We conclude that:

There can not exist an algorithm, that can identify a general computational process based upon the input/output relation.

We conclude that if we use a cipher that includes a general computational process, and keep all construction parameters of that process secret, the cryptanalyst will face a problem which he will be unable to solve.

The above proof supply us with an answer to proposition (C1). It also show that, for the HCIA cipher, the functions that solve/break the HCIA cipher is indeed "not on the black spots that the cryptanalyst can reach", and so the model of Figure 5 is a valid description of the HCIA cipher, at least in a theoretical setting where we ignore that the computational length is limited and that the memory is limited. The two models are equivalent so this is, however, to be expected.

- -

The Theorem of H. G. Rice.

In 1953 H. G. Rice published the famous theorem that bears his name [132]. Rice looked upon computation sequences, and applied a test function, a property, to the output. He then classified the test functions in this way:

A property is called a trivial property, if it is always true or false independent of what computational sequence it is applied to.
Else a property is called a non-trivial property, and it will be true for some computational sequences, and false for others.

Rice then state that:

Any non-trivial property of a Turing Machine is undecidable.

... so that there cannot exist any algorithm, method, or computation that may answer for which computations the property holds true, and for which the property is false. We see that this point of view increase confidence in the HCIA system, as long as we stay within the theoretical model. The cryptanalyst remain completely helpless. No property, what so ever, can be obtained.

There can be no algorithm for algorithms

We see that proposition (A1) is false, as a general method would require an algorithm of some kind, or at least the possible existence of an algorithm. We conclude that the alternative option (A2) must hold. (G3) is the hypothesis of a one-way function. Here, where we are not limited to a static or invariant block function, we may conclude that a secure cipher do indeed exist, but for the moment we leave open how it should be constructed.

For systems that are slightly simpler than the HCIA system, we may still find that some properties of the system remains uncomputable, while others may be computable:

"Regular languages are sufficiently simple that their properties may be determined by finite computational procedures. Properties of context-free and more complicated languages are, however, often not computable by finite means. Thus, for example, the minimal grammars for such languages (whose sizes would provide analogues of the regular language complexity) cannot in general be found by finite computations. Moreover, for context-sensitive languages and general languages, even quantities such as entropy are formally non-computable."
[ch 7, 144].

- -

The Secret Algorithm

We now make a few notes on the subject that the "actual" algorithm, the algorithm that HCIA generates, **is a secret**, that we don't reveal to the cryptanalyst. The possibility to obtain a general solution, that may work independent of the exact sequence of operations that the the HCIA perform, was concluded to be nil. The next logical step will then be to try to obtain the algorithm-instance, and then attack this algorithm using conventional methods.

When the cryptanalyst have successfully broken a cipher he may have gained a thorough knowledge of the cipher. By a "successful break" we mean that the key of the cipher, or equivalent information, has been recovered to an extent that the cryptanalyst may read encrypted information at will. It is, of course, possible to attack a cipher even if some inner workings of the cipher system is, a priori, unknown to the cryptanalyst. But we may assume that there cannot exist a successful break of a cipher, which still have (partially) unknown building blocks.

According to the Linguistic Complementarity (above) we understand that obtaining a complete description of the generated algorithm will not be an easy task:

- 1) The cryptanalyst's first option is to work in the language that is defined by the list of the HCIA-operations (this is non-secret information). Due to the level of complexity that is easily included in the operations, as in the example implementation contained in this report, it is likely that this will be too difficult for the cryptanalyst. We also know that it is fundamentally difficult to obtain a complete description of the cipher. An extremely difficult problem will be to exploit the round-off that will occur due to that a real implementation will use a finite-size memory.

But complete, or almost complete, the description must be:

- 2) If the cryptanalyst simplifies the the description, and don't include all possible details of the implementation at the same time in his description, then the simplified description will be more general, and the description of the HCIA will now be equivalent to the theoretical Turing machine model of computation, that is proven to be mathematically impossible to attack.

We see that even if the theoretical Turing machine model of the HCIA cipher is merely an approximation -- due to limited non-infinite memory in any actual physical implementation -- the cryptanalyst may be forced to work in this model anyway.

We may compare with ordinary algorithm-ciphers, where the connection between the cipher algorithm and the cryptanalyst through the intermediate description, is not at all as quarrelsome. This is due to that the language of algorithm-ciphers are simpler. For the HCIA, that use a maximum-complexity language, the difference between the actual implementation (the interpretation of the cipher) and its

corresponding best partial description, becomes a significant and important obstacle for the cryptanalyst.

The second option for the cryptanalyst is to exploit the possible transcendability of the linguistic complementarity, and search for a complete description on a higher meta-level. We conclude that the cryptanalyst, in order to obtain a description, must work on an even higher level than the cipher itself. Well, there is always the option that the "higher meta-level" might not exist.... If it exist it should be higher than the the recursive languages (the language of the Turing machines). Unfortunately, any language beyond the recursive languages will no longer be recursive, the methods will not be computable, the questions undecidable.

We conclude that proposition (B1) must be false, as there are several good arguments on this. (B3) must also be false. We also have a strong argument for that (C3) is false. Proposition (E1) (and E2) are true, except from that there will be no successful cryptanalysis, as we have seen above.

We also have material on the proposition (G2). We cannot build more complex systems than the HCIA ciphers. The reason for this is that the HCIA uses the computational process directly in its definition. An even more general, and complex, solution will be beyond the recursive languages, i.e. that they cannot be implemented by any effective means. Hence there can only be the HCIA-ciphers beyond the algorithm-ciphers.

The Classification of Ciphers

The discussion above now enable us to classify all encryption according to their properties. The list below is complete in the sense that every secret-key cipher must fit into one group, but, as evident, only a partial description of each group will be given!

Compare the classification of ciphers with the classification of formal languages [144 sec 1].

Class 1: Hand-Ciphers

Implementation:	Paper and pen
Execution:	Directly
Internal storage:	One letter
Security Philosophy:	Secret System
Cryptanalysis:	Known plaintext

With a hand-cipher, sometimes called a tactical code, the plaintext is encrypted manually. Paper and pen is most often used. Most hand-ciphers encrypt a single letter at a time. Almost all ciphers in this class have low security and can easily be broken using methods published in open literature [112], [118], [127], [128]. Tables and alphabets used for encryption are often equivalent with the key of the cipher.

Example: Simple substitution ciphers. See [129] for a discussion on codebooks.

Note: One-Time-Pad ciphers [135 Ch 1.2.4 p 13-15], [126] is not included in this class.

David Kahn provide many valuble details in a historical context [120].

Class 2: Algorithm Ciphers.

Implementation:	Algorithm
Execution:	Time-invariant
Internal storage:	Small block
Security Philosophy:	Secret Key
Cryptanalysis:	Statistical Attack

An algorithm cipher has a time-invariant algorithm [143 ch 3.1 p 23], [122 Ch 1.1 p 4-6], [107 ch 1.1] which specifies how the cipher works. The cipher uses an external key, a binary string, according to specifications in the algorithm. It is assumed that internal storage of the algorithm is of time-invariant size.

Blockciphers [135 Ch 8.1 p 154-165] and streamciphers [135 Ch 8.3 p 168-176] are examples of ciphers in this class [135 Ch 8.4 p 176-177], [143 ch 3.5(c) p 35]. The internal storage of a streamcipher consists of the current state of the internal key generator. This could be implemented by shift registers. The internal storage of a blockcipher consists of a specified number of bits, often the same as the blocksize.

Algorithm ciphers generally provides greater resistance against cryptanalysis compared to hand ciphers. Modern methods may render older systems insecure [103][].

Class 3: HCIA-Ciphers

Implementation:	Interpretation
Execution:	Self-Updating
Internal storage:	Substantial
Security Philosophy:	New system every time
Cryptanalysis:	Description Problem

A non-algorithmic cipher clearly cannot be implemented by any algorithm. Instead, a non-algorithmic cipher is an interpretation [124 sec 3 p 329]. The computer program, which implements the cipher, contains specifications for the interpretation. What the interpretation would generate during execution (the "real" cipher) is unknown prior to this execution.

A cryptanalytic approach towards class 3 ciphers immediately reveals that no ordinary analytic tool may work. Unconditionally secure ciphers [135 Ch 1.2.4 p 13-15], [126] belong to this class.

Class 4: Noncomputable Ciphers.

Implementation:	Unpractical
Execution:	Terminates?
Internal storage:	Limited?
Security Philosophy:	-----
Cryptanalysis:	??????

Ciphers in class 4 are noncomputable [123]. It means that the cipher, or some part therein, cannot be computed by any effective means. All computable ciphers belong to one of the classes 1, 2 or 3. All ciphers thus belongs to one of the classes 1, 2, 3 or 4. In this paper we will not attempt to investigate this cipher class further.

Historical Context

When we compare different ciphers, we see that the cipher security increases, as we move towards a higher cipher class. The two most important properties are memory and variation. If the cipher is based upon a time-invariant function, and has only little internal memory, it could be easy to attack. But if the cipher has much memory, and if the rules of calculation are varying, it get increasingly more difficult to find a systematic method that makes the cipher reveal its secrets [143 ch 3.4(a)p 30 lines 19-21] [139 p24 lines 23-25]

In Encryption, Memory is **EVERYTHING!**

Examples may be taken from a historical context. In [110 ch 2 p 35-91] the development of the ECM MARK II cipher machine is described. A machine proposed by E. H. Hebern [115] was found to be weak [110 p 47] due to too infrequent and too regular rotor stepping. This was remedied by an invention by Frank Rowlett [110 p78 last paragraph]. ECM MARK II contains three sets of rotors, the first two sets being a pseudo random number generator controlling the stepping of the last set, that encrypts the plain text. The security of ECM MARK II may be compared to the security of the German Enigma [131][117], which had regular rotor stepping. See also the patent by Warring [162].

Codebreaking: In The Real World

Now shall we address some more philosophical aspects of cryptanalysis, and also obtain some of the strongest arguments for the correctness of the HCIA-system.

We have seen above that, if cryptanalysis of a HCIA-powered system is possible, this will be limited to individual networks and messages, as the generated algorithm is used only once. This limits the damage would there an occasional break. (There will not be any such break, but if! if!)

Previously we used diagonalisation, invented by Georg Cantor. The other important tool of complexity theory, **reduction**, now come into play. By using reduction we may reduce one problem onto another, and so obtain important information about the relative difficulty of tasks.

We note that the general problem, of investigating the properties of software, has been extensively studied in the software industry. We may compare an cryptanalyst, attempting cryptanalysis on a HCIA-cipher, and a software engineer, struggling with debugging a problematic software.

The cryptanalyst would be investigating the properties of a universal machine that reads a string $\{M,K\}$ and outputs a string $\{C\}$. We assume that the string $\{M\}$ is known by the cryptanalyst, who attempts to find a key $\{K\}$ such that $\{C\} = \{C_0\}$. The software engineer will be investigating the behaviour of an all-purpose computer executing a software $\{P\}$ with input parameters $\{x\}$. The engineer observes the output $\{Y\}$, and tries to find an input $\{x\}$ that makes the computation behave in some specified way $\{Y\} = \{Y_0\}$.

The comparison $\{C\} = \text{computation}(\{M,K\});$
 $\{Y\} = \text{computation}(\{P,x\})$ gives that breaking the proposed cipher will be at least as hard as debugging software.

Clearly, the cryptanalyst will not be allowed to inspect the software, single-step using a debugger, or inspect the internal state of the memory, tools that is essential for the success or failure of the software engineer.

We may conclude that if there is a tool, method, process, or algorithm, that is strong enough to break any HCIA encryption system, then this tool must also be strong enough to be able to detect, or obtain, the problematic input {x} that makes the investigated software {P} go "bang". A few seconds of thought -- that the HCIA system generates new software on each encryption -- show that this cannot occur as an occasional accident.

We thus face a generic tool, that we may use to systematically remove bugs from software. We note especially that actual software implementations are limited, by a limited memory and a limited computational sequence, exactly as the HCIA system is. So for this reduction, there is no theory-reality round-off. The drawback is that we cannot know, possibly we will never know, if such a tool can be built.

But if it can be built, it will once and forever, and for every type of software, remove all programmers mistakes, buggs, and strange error messages. We now follow this line of thought: Do such a too exist today? Can it exist in future??

First, the tool do not exist today. Period. The world wide cost of producing software is so enormous, and the issue so well monitored, by so many journalists and others, that it is ridiculous to propose that a universal bug-remove tool could exist today. Further, it cannot be kept secret by any "agencies":

- 1) Cryptology is the art of secrecy, where we have the strange (ridiculous?) situation that there is both "public" and "secret" research. This is not the case for software engineering, where all tools, tricks, and methods are open, and have always been open. This was one of our main objectives with the HCIA in the first place -- do not base the design upon a construction where there might exist powerful unpublished research and attacks...
- 2) We know, unfortunately, that the "secret" military sector is hurt by the same software buggs that is so difficult to remove from commercial software. A short thought -- and we remember when military aircraft, space missions, computerised weapon systems have misbehaved in dangerous and unpredictable ways.

Based upon the comparison, above, we can conclude that the difficulty of removing (detecting) the last bug in a commercial software is at least as difficult as a successful cryptanalysis of the HCIA system.

From a philosophical position we may choose not to make predictions about the future. We then conclude that, today, the HCIA system remain secure.

If we allow ourselves to make a prediction about the future --
it is likely that we will forever live with the fact that large and
complex computer systems occasionally malfunction, and that the
HCIA system forever remains secure.

The author wish to add that he would, however, prefer if the buggs that destroy things and kills people could be removed from software, even if this would imply that the HCIA system could be broken.

- -

Other Attacks

A cipher will clearly not be super-secure merely if it includes a copy of the software found in this report. So what can we expect the HCIA to do?

Following H. G. Rice we call an attack "trivial" if it can be mounted on a HCIA-powered cipher independent of the numbers input to the HCIA system. An attack is called non-trivial if it apply only to some input strings to the HCIA system, and not to others.

There can be no non-trivial attacks

So the HCIA may be used as a pseudo-random number source, that is impossible to break if only partial information about this reach the cryptanalyst. But all other trivial attacks, that may be mounted on the cipher, must be managed and taken care of by the cryptographer himself. The important difference with the HCIA is thus:

- 1) There can be no high-complex attacks.
- 2) There can be no attacks based upon advanced, possibly secret, research.
- 3) The HCIA cipher cannot protect against implementation mistakes, key distribution problems, electrical emission from encryption equipment, error in red-black separation, "practical cryptanalysis" (theft, threat, spies, bribes, and other bad behaviour), the operator may sell the keys [129], and so on.

Proposition (D1) is true.

The Mathematical Proof

Suppose that we, as a thought experiment, make a formal description of a HCIA system:

Formal description of HCIA encryption system...

In the next step we actually prove that the HCIA system is mathematically secure.

Theorem: The HCIA cipher cannot be broken

Proof:

But this cannot be the case -- a mathematical proof is "true" only if it is syntactically true -- it must be true in any interpretation. But this cannot work for the HCIA cipher. The problem is that

- (a) we cannot describe the HCIA system completely in a formal language
- and
- (b) the security depend on the interpretation (the numbers processed), and so cannot be independent of this.

So there may never be any proof of security.

We conclude that if -- if -- we could mathematically prove that the HCIA system is secure, we must have obtained a description of the HCIA system, and hence broken the system!

We conclude that no such description can ever be found, and consequently the theorem is true!

Speed/Security Paradox

Encryption solutions exist of varying quality. Each different cipher have different computational costs and obtain different security levels. Weak ciphers are always a possibility, and whether they are slow or fast is of no interest. Extremely secure systems also exist. These systems tend to be very slow. For a cipher-algorithm, the security can be increased only by adding more complications, that cost processing time or hardware layout size. This is also no surprise, as we may simply concatenate different, preferably independent [116], cipher algorithms to increase security for a speed penalty.

Encryption security is independent of the computational cost

Proof: Assume that it is impossible to increase security without a speed penalty. Assume that a (fast) HCIA system is being used. Suppose, as an example only, that we use the HCIA-operations that is included in this report. We assign the implementation a specific security level.

- 1) If we increase the number of operations that the HCIA run for each byte or block of plaintext, the complexity and hence the security will increase(*).
- 2) If we increase the work performed in each HCIA-operation, the security will increase. Possibly alternative (1) is more effective than (2).
- 3) If we increase the number of HCIA-operations, the complexity will also increase, and consequently will the security be higher. For our example cipher may we double the number of operations from 64 to 128.

Using (3) we may increase or improve a HCIA system. The change can be performed at no or at very low cost. Especially for a software solution, the cost of a few bytes larger software is nil. Using (3) the execution time for the cipher need not need to change. We merely should assure that the average execution time for the new operations is the same as for the old previously existing operations.

We see that we can improve the security of a HCIA system without any speed penalty. If this is true, for this particular cipher, the proposition above must generally be false. We conclude that the security of encryption cannot be correlated with the corresponding computational cost.

(*) We know that increasing the linguistic complexity of the cipher increase security, and if the recursive languages are reached, the system is (absolutely) secure.

References and Further Reading

Due to editing in the text, the following references are not quoted:

101, 104, 106, 108, 111, 119, 125, 136, 138, and 142.

[101]

Advanced Encryption Standard
Conference

[102]

Aho, Alfred V.; Sethi, Ravi; and Ullman, Jeffrey D.
"Compilers" "Principles, Techniques, and Tools"
Addison-Wesley series in Computer Science 1986
ISBN 0-201-10194-7 12 MA 9695

[103]

Andelman, Dov
"Maximum Likelihood Estimation Applied to Cryptanalysis"
Ph.D. dissertation
Electrical Engineering Dpt.
Stanford University
Stanford CA, Dec 1979
(UMI Dissertation Services <http://www.umi.com>)

[104]

Anderson, Ross
"Non-cryptographic Ways of Loosing Information"

[105]

Beckman, Bengt
"Svenska kryptobedrifter"
Albert Bonniers Förlag 1996
ISBN 91-0-056229-7

[106]

Bianco, Mark E. and Reed, Dana A.
"Encryption System Based on Chaos Theory"
United States Patent 5.048.086
Sep 10, 1991

[107]

Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.
"Introduction to Algorithms",
McGraw-Hill, 1989
ISBN 0-07-013143-0

[108]

Daemen, Joan
"Limitations of the Even-Mansour Construction",
Lecture Notes in Computer Science,
Proceedings Asiacrypt 91', page 495-498, Springer-Verlag 1992

- [109]
Davis, M.
"The Undecidable",
Raven Press, Hewlett, New York, 1965
- [110]
Deavours, Cipher A. and Kruh, Louis
"Machine Cryptography and Modern Cryptanalysis"
Artech House, Inc., 1985
ISBN 0-89006-161-0
- [111]
Diffie, W. and Hellman, M.E.
"Privacy and Authentication: An Introduction to Cryptography"
in
Proceedings of IEEE, vol 67, No. 3 March 1979
Pages 397-427
- [112]
Edwards, Daniel James
"OCAS: On-line Cryptanalytic Aid System",
Master's thesis
NTIS Accession number: AD-633 678
Report no: MAC-TR-27,
MIT Project MAC,
Massachusetts Inst of Tech, Cambridge
May 1966.
- [113]
Gödel, Kurt
"Über formal unentscheidbare Sätze der Principia mathematica
und verwandter Systeme I",
Monatshefte für Mathematik und Physik 38,
pp 173-198, 1931.
(Received November 17, 1930)
- [114]
Gödel, Kurt
"On undecidable propositions of formal mathematical systems",
1934
Printed in [109]
- [115]
Hebern, E. H.
"Electric Coding Machine"
US patent 1,510,441 Sept 30, 1924

- [116]
Herlestam, Tore
"Kryptoanalytiska synpunkter på några aktuella
krypteringsfunktioner"
Fst/TSA/Text TH771209
- [117]
Hinsley, Francis Harry and Stripp, Alan (editors)
"Codebreakers -- The Inside Story of Bletchley Park"
Oxford University Press 1994
ISBN 0-19-285304-X
- [118]
Howe, Walt.
"Basic Cryptanalysis, FM 34-40-2"
U.S. Army 1990
Classical Crypto Books
P.O Box 1013, 47 Wiley Hill Road,
Londonderry NH 03053-1013 U.S.A.
- [119]
Ingemarsson, Ingemar
"Encryption With Randomly Chosen Unknown Functions"
1977 IEEE International Symposium on Information Theory
pp 44 (Abstracts Only)
IEEE Cat Nr 77CH 1277-3 IT
Library of Congress no. 79-173358
- [120]
Kahn, David
"The Codebreakers: The Story of Secret Writing",
New York: McMillan, 1967
- [121]
Kleene, Stephen C.
"Turing's Analysis of Computability
and Major Applications of It"
pages 17-54 in
Herken, Rolf (ed)
"The Universal Turing Machine -- A Half-Century Survey"
Oxford University Press, 1988
ISBN 0-19 853 741-7
(See [109])
- [122]
Knuth, Donald E.
"The Art of Computer Programming"
"Volume 1/Fundamental Algorithms"
Second Edition
Addison-Wesley Publishing Company 1973
ISBN 0-201-03809-9

- [123]
Löfgren, Lars
"Computability",
Systems & Control Encyclopedia,
Cambridge: Pergamon Press, 1987
- [124]
Löfgren, Lars
"Autology",
Systems & Control Encyclopedia,
Cambridge: Pergamon Press, pp 326-333, 1987
- [125]
Matthews, Robert
"On the Derivation of a "Chaotic" Encryption Algorithm"
Cryptologia, Vol XIII Number 1, pages 29-42, January 1989
- [126]
Mauborgne, Joseph
The OTP system.
- [127]
Navy Department,
Office of Chief of Naval Operations
Code and Signal Section
"Elementary Cipher Solution"
Cryptologia, Vol XXII Number 2, pages 102-120, April 1998
- [128]
Nichols, Randall, K
"Classical Cryptography Course"
Volume II
Aegean Park Press 1996
ISBN 0-89412-264-9
- [129]
Nilsson, Lars Axel
"Krypto"
Aldusserien 289
Bokförlaget Aldus/Bonniers Stockholm 1970
- [130]
Papadimitriou, Christos H.
"Computational Complexity"
Addison-Wesley Publishing Company 1995
ISBN 0-201-53082-1

- [131] Rejewski, Marian
"How Polish mathematicians deciphered the Enigma"
Annals of the History of Computing
Vol 3 No 3 July 1981, pp 213-229
translated by American Federation of
Information Processing from
"Jak matematycy polscy rozszyfrowali Enigme"
Annals of the Polih Mathematical Society, Series II
Wiadomosci Matematyczne, Volume 23, 1980, 1-28
- [132] Rice, H. G.
"Classes of recursively enumerable sets
and their decision problems"
Trans. Amer. Soc. vol 74 pp 358-366, 1953
- [133] Ritter, Terry,
Cipher by Ritter
- [134] Rodin, Gunnar
"Metoder för kryptering av datorlagrade data"
(Statskontoret 1972-05-10)
Institutionen för Informatonsbehandling - ADB
Kungl. Tekniska Högskolan, Stockholm, 1972
- [135] Schneier, Bruce
"Applied Cryptography -- Protocols, Algorithms,
and Source Code in C"
John Wiley & Sons, Inc., 1994
ISBN 0-471-59756-2
- [136] Schneier, Bruce
The Blowfish algorithm
- [137] Shannon, C.E.
"Communication Theory of Secrecy Systems",
Bell System Tech. J., Vol 28, 1949, pp. 656-715
- [138] Stewart, Ian
"Does god play dice? The Mathematics of Chaos"
Penguin Books 1990
ISBN 0-14-012501-9

- [139]
"Kryptering i ADB-system.
Praktisk hjälpreda för beslutsfattare och systemerare"
SIS teknisk rapport 312 Utgåva 1 November 1985
Riksdataförbundet
SIS Standardiseringsgrupp
Sårbarhetsberedningen
ISBN 91-7162-191-1
- [140]
Tarski, Alfred
"Der Wahrheitsbegriff in den formalisierten Sprachen",
Studia Philosophica, vol. 1 pp 261-405, 1935.
- [141]
Turing, Allan Mathison
"On computable numbers, with an application to
the Entscheidungsproblem"
Proc. Lond. Math. Soc. 42, 230-265 (1936-1937);
received May 25, 1936,
Appendix added August 28; read November 12, 1936;
A correction, *ibid.*, 43 pp 544-546, 1937
- [142]
Weihrauch, Klaus
"Computability"
#9 EATCS monographs on Theoretical Computer Science
Springer-Verlag 1987
ISBN 3-540-13721-1
ISBN 0-387-13721-1
- [143]
Widman, Kjell-Ove in
Christoffersson, Per,
Fåk, Viiveke (editor)
"Crypto Users Handbook"
North-Holland 1988
ISBN 0-444 70484-1
- [144]
Wolfram, Stephen
"Computation Theory of Cellular Automata"
Communications in Mathematical Physics vol 96
pages 15-57,
November 1984

- [145]
Zischang, Thilo
"Combinatorial Properties of Basic Encryption Operations"
Advances in Cryptology-Eurocrypt '97, pp 14-26,
Walter Fumy (ed.)
Springer-Verlag, LNCS # 1233, Berlin 1997
ISBN 3-540-62975-0
- [146]
Dauben, Joseph Warren
"Georg Cantor"
"His Mathematics and Philosophy of the Infinite"
Princeton University Press 1990
Princeton, New Jersey 08540
ISBN 0-691-08583-8 (cloth)
ISBN 0-691-02447-2 (paper)
- [147]
Cohen, P. J.
"The Independence of the Continuum Hypothesis I"
Proceedings of the National Academy of Sciences,
50 (1963)pp 1143-1148
U.S.A.
- [148]
Cohen, P. J.
"The Independence of the Continuum Hypothesis II"
Proceedings of the National Academy of Sciences
51 (1964), pp 105-110
U.S.A.
- [149]
Reid, Constance
"Hilbert"
Springer-Verlag New York 1996
SPIN 10524543
ISBN 0-387-94674-8
Originally published Springer-Verlag Berlin New York 1970
ISBN 0-387-94674-8
- [150]
Löfgren, Lars
"Automata och Formella Språk"
Kompendium Teoretisk Automatik
Lunds Universitet
1987

- [152] IEEE 802.11 Wireless LAN
<http://www.ieee802.org/11>
<http://standards.ieee.org/getieee802/>
- [153] Borisov, Nikita; Goldberg, Ian; Wagner, David
wep@isaac.cs.berkeley.edu
"Security of the WEP algorithm"
<http://www.isaac.cs.berkeley.edu/isaac/wep-faq.html>
<http://www.isaac.cs.berkeley.edu/isaac/wep-slides.pdf>
<http://slashdot.org/articles/01/02/15/1745204.shtml>
- [154] "Wireless Research"
"802.11 Security Vulnerabilities"
A number of security vulnerabilities have been identified by ourselves and other researchers. Links to the original papers/presentations are provided below in chronological order:
<http://www.cs.umd.edu/~waa/wireless.html>
- [155] Curtin, Matt cmcurtin@interhack.net
"Snake Oil Warning Signs: Encryption Software to Avoid"
Copyright ©1996-1998 April 10, 1998
<http://www.interhack.net/people/cmcurtin/snake-oil-faq.html>
- [156] Herlestam, Tore
"Kryptering-för säkerhets skull"
Tidningen Elteknik med aktuell elektronik
1979:14
- [157] Vretblad, Anders (Ed.)
"FESTSCHRIFT in honour of Lennart Carleson and Yngve Domar"
Proceedings of a Conference at the Department of Mathematics,
Uppsala University, May 1993
Acta Universitatis Upsaliensis "C" Org. & Hist. #58
Printed by Gotab, Stockholm 1995
Distribution: Almqvist & Wiksell Internat., Stockholm, Sweden
ISSN 0502-7454
ISBN 91-554-3574-2
- [158] Cantor, Georg
"Über eine elementare Frage der Mannigfaltigkeitslehre"
Jahresbericht der Deutschen Mathematiker-Vereinigung I;
pp 75-78.

- [159]
Jan von Plato
"Creating Modern Probability"
Cambridge University Press 1994
ISBN 0-521-44403-9 hardback
ISBN 0-521-59735-8 paperback
- [160]
Bauer, Friedrich L. (Prof. Dr. Dr. h. c. mult.)
"Decryped Secrets"
"Methods and Maxims of Cryptology"
Springer-Verlag Berlin Heidelberg New York 1997
ISBN 3-540-60418-9
- [161]
Kerckhoffs, Auguste
"La cryptographie militaire"
1883
- [162]
Warring, S. E.; Skärholmen
"Anordning att alstra en serie digitala signaler"
Pat. SE-332 257
- [163]
Dömstedt. Bo and Stenfeldt, Mats.
"Processing method and apparatus for converting
information from a first format into a second format"
Patent Applications PCT/SE99/01740; EP 98118910.3,
Lateca Computer N.V.
- [164]
Cypress Microsystems Inc.
"PSoC Designer: Assembly Language User Guide v. 2.0"
October 2002
<http://www.cypressmicro.com/pdf/AssemblerUserGuide.pdf>
- [165]
Microchip
Application Note AN536
"Basic Serial EEPROM Operation"
<http://www.microchip.com/1000/suppdoc/appnote/all/an536/index.htm>
- [166]
"Crypt-X 98"
Information Security Research Centre,
Centre in Statistical Science and
Industrial Mathematics,
Queensland University of Technology.
<http://www.isrc.qut.edu.au/resource/cryptx/>