Isomorphism Theorem and Cryptology

R. L. de Carvalho and F. L. de Mello

Abstract— **This paper presents a Theory of Computation study based on recursive functions computability and innovates by performing parallels to relevant themes of Cryptography. Hence, it is presented the Hennie's notion of "abstract family of algorithms" (AFA, for short) according to the authors' understanding, and also more judicious theorems demonstrations, many times completely different from those ones available in literature. The main issue is the Isomorphism Theorem which supports the Church-Turing Thesis and provides a connection between Cryptology and Linguistics.**

Keywords— **Algorithm, Recursive functions, Church-Turing Thesis, Fixed point theorem, Recursion theorem, Isomorphism theorem.**

I. Introduction

THE CODIFICATION is based on conversion rules whose objective is to transform a piece of information, a message, into a new representation of the same information. Under the Cryptology point of view, this concept remains valid, but it is added to it a strong constraint associated to the intention of keeping the message content restricted to an entity group, and obscured to everyone else. Thus, the informative content representation is transformed by an algorithm, producing a symbol sequence which belongs to the new depiction universe. Those transformation procedures are known by the Theory of Computation as Transductive Formal Systems [7].

Algorithms that convert a sequence of symbols into a new one, preserving the informative content, are known as compilers and interpreters. Besides the existence of important differences between compilers and interpreters, both of them have the responsibility of performing a language translation. Usually, this kind of translation is done from a high-level language to a low-level one. However, there is no obstacle against a translation performed on the other way round, or even from high-level to high-level, or from low-level to low-level.

Similarly, a sequence of symbols from a language plaintext is converted by a encoding algorithm into a new sequence from a cryptographic language. By this reason, it possible to be more specific by asserting that cryptographic algorithms translate a plaintext message to other languages such as Triple-DES, Blowfish, SEAL, MD5, among others. This approach is unconventional in Cryptology. Nevertheless, the existence of terms such as alphabet, communication, dictionary, corpus and corpora, available not only on Cryptography studies, but also on general Linguistics studies and on Theory of Computation, suggests an obtainable intersection regions among these knowledge areas.

R. L. de Carvalho (Ph.D.), Witty Group leader for artificial intelligence and knowledge visualization fields, rlins@globo.com

F. L. de Mello (D.Sc.), Assistant Professor at Electronics and Computation Engineering Department from Polytechnic School at Federal University of Rio de Janeiro, fmello@del.ufrj.br

Invited Paper

On Theory of Computation, the Church-Turing Thesis states the direct relationship between algorithms and languages. It provides a correlation between the act of calculating and the algorithms materialization, that is, the computer programs. The calculus is the execution of methodic sequence actions, and the programs representation is provided by the language.

Thus, this article aims to use Theory of Computation concepts so as to produce a deeper comprehension about the algorithms and the objects to be represented. This approach provides a rigorous understanding about the subject and suggests a correlation between Cryptology and Linguistics.

II. Church-Turing Thesis

The Church-Turing Thesis, shown in Fig. 1, is not a theorem but an epistemic result which acceptance is almost universal. The intuitive concept of computable is associated to a class of arithmetic functions called recursive functions [1]. It can be reasoned that if a hypothesis cannot be directly proved, then maybe it can be refutated. Consequently, in order to deny the thesis it is sufficient to find out just one procedure that cannot be demonstratively computed by a Turing Machine. This procedure has not been found so far, and more over, because there are a considerable number of favorable experimental data, researches tend to accept the thesis. In addition, several attempts to specify the algorithm concept resulted into formalisms that can be demonstrated as equivalent to the Turing Machine.

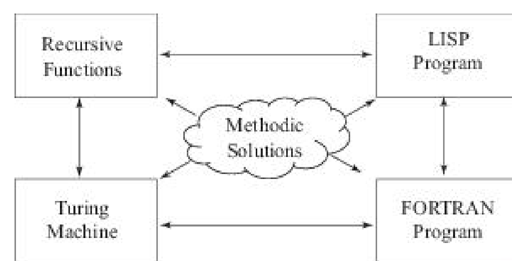

Figure 1. Illustrative scheme from Church-Turing Thesis suggesting that all formalisms to define an algorithm are supposed to be equivalent among themselves.

Therefore, the Church-Turing Thesis is a hypothesis on the mechanical nature of the act of calculating, describing a direct relationship to the computer, and to the different types of algorithms that can be executed by a machine. Thus, every function considered to be systematic can be computed by a Turing Machine. Any kind of programs can be translated to a Turing Machine, and also, any Turing Machine can be translated to a programming language. Consequently, any ordinary programming language is sufficient to represent any algorithm, whatever is its purpose.

III. Algorithms

An algorithm a description of a function calculus or evaluation performed in a systematic way. The main elements associated to this idea are enumerated as follows:

1) *Finite size instruction set*: an algorithm must be described by a language into a finite fashion. Assuming an enumerable alphabet, the algorithm must be composed by a finite string over this alphabet.
2) *Algorithm domain*: composed by data, the set objects processed by an algorithm, for instance, a chain of symbols, natural numbers, etc.
3) *Computer agent*: the description computation results into a well defined sequence of operations or steps, that depends on an agent associated to the algorithm. This agent must be deterministic, that is, he should react to the algorithm instructions forever in the same way. For each input, the algorithm must have always the same behavior: if it halts, it will have always the same output; if it does not halt, it diverges.
4) *Facilities to execute, store and retrieve steps*: the intuitive notion of memory emerges as an agent resource. The maximum size of the memory, and consequently the input size of the algorithm, is an aspect to be taken for each computer agent. However, once this limit can be indefinitely extended, its existence is not considered.
5) *Agent capability*: by using a limited set of skills, the agent must be capable of computing any algorithm.
6) *The end of computation*: once all instructions have been executed, the agent provides the appropriate results, according to the input arguments of the domain. This does not mean that the algorithm should halt to any domain input. For instance, the natural numbers division can indefinitely operate if the dividend is not divisible by the divisor.

One of the first models of abstract machine, as an attempt to define an algorithm, was the Turing Machine. The following excerpt is a abridgement of Alan Turing understanding of what is a computer [12]. The reader must be aware that Turing wrote this text before the invention of the machine called computer, but it is really fascinating how it is still acceptable and precise.

Computing is normally done by writing certain symbols on paper[1]*. We may suppose this paper is divided into squares like a child's arithmetic book. [...] it will be agreed that two-dimensional character of paper is no essential of computing. [...] I shall also suppose that the number of symbols which may be printed is finite. [...] The behavior of the computer at any moment is determined by the symbols which He is observing, and his "'state of mind"' at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite. [...] Let us imagine the operations performed by the computer to split up into "'simple operations"' which are so elementary that it is not easy to imagine them further divided. Every such operation consists of some change of the physical system consisting of the computer and his tape. [...] We may suppose that in a simple operation not more than one symbol is altered. [...] Besides these changes of symbols, the simple operations must include changes of distribution of observed squares. The new observed squares must be immediately recognizable by the computer. I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed square does not exceed a certain fixed amount. [...] The operation actually performed is determined, as has been suggested, by the state of mind of the computer and the observed symbols. In particular, they determine the state of mind of the computer after the operation is carried out.*

[1] This is a 1937 text.

***Definition** 1:* A Turing Machine is a quintuple $M = \langle k, \Sigma, \delta, s, F\rangle$ where [4]:

- k is the finite set of STATES;
- Σ is an ALPHABET which contains symbols $\rhd$ and $\sqcup$, but doesn't contains $\rightarrow$ and $\leftarrow$;
- $s \in k$ is the INITIAL STATE;
- $F \subseteq k$ is the set of HALTING STATES;
- δ is a TRANSITION FUNCTION of $k \times \Sigma$ where: (a)
  1) for all $q \in (k - F)$, if $\delta(q, \rhd) = (p, b)$, then $b = \rightarrow$;
  2) for all $q \in (k - F)$ and $a \in \Sigma$, $a \neq \rhd$, if $\delta(q, a) = (p, b)$, then $b \neq \rhd$.

□

IV. Recursive Functions

One usual approach to define a mathematical function is the *recursive definition*: some initial function values are defined and the other ones are computed based on the priors. It is essential to understand the recursive functions computational model that is presented at this section. On the other hand, the reader who is acquainted with this subject may proceed to section VI without prejudice to the understanding of this work.

The recursive definition method is used to characterize the primitive recursive function class. First, some initial functions are defined, whose simplicity suggests their unconditional computability. These functions are described as follows [1][4]:

Successor: $\underline{suc} : \mathbb{N} \rightarrow \mathbb{N}$, so that for all $x \in \mathbb{N}$

$$\underline{suc}(x) = x + 1$$

Zero: $\underline{zero} : \mathbb{N} \rightarrow \mathbb{N}$, so that for all $x \in \mathbb{N}$

$$\underline{zero}(x) = 0$$

Projection: for each $n > 0$ and each $1 \leq i \leq n$, $\underline{pr}_i^n : \mathbb{N}^n \rightarrow \mathbb{N}$, so that for all $\underline{x}_n =< x_1, x_2, \cdots, x_n >\in \mathbb{N}^n$

$$\underline{pr}_i^n(\underline{x}_n) = x_i$$

Subsequently, it is necessary to define a procedure responsible for describing functions by using the previous ones as support, on the early case, the initial functions. Assume the functions $f : \mathbb{N}^m \rightarrow \mathbb{N}$ and $g_1, \cdots, g_m : \mathbb{N}^n \rightarrow \mathbb{N}$

exists. The composition h of f and $g_1, \cdots, g_m$ is the function $h : \mathbb{N}^n \to \mathbb{N}$, defined by [1][4]:

$$h(x_1, \cdots, x_n) = f(g_1(x_1, \cdots, x_n), \cdots, g_m(x_1, \cdots, x_n))$$

The composition of a function f with other function g is usually denoted by $f \circ g$.

At last, let $f : \mathbb{N}^n \to \mathbb{N}$ and $g : \mathbb{N}^{n+2} \to \mathbb{N}$. It is said that $h : \mathbb{N}^{n+1} \to \mathbb{N}$ is defined by a primitive recursion if the h values are obtained by [1][4]:

$$\begin{array}{lcl} h(0, x_1, \cdots, x_n) & = & f(x_1, \cdots, x_n) \\ h(y+1, x_1, \cdots, x_n) & = & g(y, h(y, x_1, \cdots, x_n), x_1, \cdots, x_n) \end{array}$$

The sum of natural numbers, for instance, can be defined as follows:

$$\begin{array}{lcl} 0 + x & = & x \\ (y+1) + x & = & (y+x) + 1 \end{array}$$

So, by taking: $f = \underline{pr}_1^1 : \mathbb{N} \to \mathbb{N}$ and $g = \underline{suc} \circ \underline{pr}_2^3 : \mathbb{N}^3 \to \mathbb{N}$, obtained by composition of the initial function, it is possible to write:

$$\begin{array}{lcl} h(0, x) & = & f(x) = \underline{pr}_1^1(x) \\ h(y+1, x) & = & g(y, h(y, x), x) = \underline{suc}(\underline{pr}_2^3(y, h(y, x), x)) \end{array}$$

Notice that for all x and all y, $h(x, y) = x + y$, therefore h is the natural numbers addition. Moreover, h was obtained by composition and primitive recursion based on the initial functions. So, to calculate the sum of two numbers, such as 3 and 5, the computation is given by:

$$\begin{array}{lcl} h(0, 5) & = & 5 \\ h(1, 5) & = & \underline{suc}(h(0, 5)) = \underline{suc}(5) = 6 \\ h(2, 5) & = & \underline{suc}(h(1, 5)) = \underline{suc}(6) = 7 \\ h(3, 5) & = & \underline{suc}(h(2, 5)) = \underline{suc}(7) = 8 \end{array}$$

Therefore, a function $f : \mathbb{N}^n \to \mathbb{N}$ is *primitive recursive* if f is a initial function or if it is obtained by using the composition and the primitive recursion based on the initial functions.

In order to prove that a function $f : \mathbb{N}^n \to \mathbb{N}$ is primitive recursive, it is sufficient to show its informal definition as the one used for the sum. Thus, for example, the product of natural numbers is defined by:

$$\begin{array}{lcl} 0.x & = & 0 \\ (y+1).x & = & y.x + x \end{array}$$

And there is no need to write

$$\begin{array}{lcl} .(0, x) & = & \underline{zero}(x) \\ .(y+1, x) & = & +(\underline{pr}_2^3(y, .(y, x), x), \underline{pr}_3^3(y, .(y, x), x)) \end{array}$$

which is the strict formal definition using primitive recursion.

Nevertheless, not all functions define by recurrent schemes are primitive recursive. As an example, assume the Ackermann function [1][4]:

$$\begin{array}{llcl} a.1 & a(0, y) & = & y + 1 \\ a.2 & a(x+1, 0) & = & a(x, 1) \\ a.3 & a(x+1, y+1) & = & a(x, a(x+1, y)) \end{array}$$

Notice that this definition uses a recursion scheme, and that for all naturals m and n it is always possible to compute $a(m, n)$. By computing $a(1, 2)$ it is obtained the value 4, but the definition of the Ackermann function is quite different from the primitive recursive definition. It is interesting to observe that the general recursion scheme may define functions that are computable for a certain instances, but eventually they diverge.

A detailed study of general recursive schemes, which are different from the recursive primitive recursion, is not the purpose of this article. The existence of computable functions defined by those schemes, and that the general recursive schemes may define computable but divergent functions, suggests that besides composition and primitive recursion, it is necessary to define an additional functional scheme. This need is fulfilled by the *minimization* scheme, responsible for producing total functions.

Let $f : \mathbb{N}^{n+1} \to \mathbb{N}$ be a total function. The $h : \mathbb{N}^n \to \mathbb{N}$ is defined by minimization of f if and only if the values of h are obtained by [13]:

$$h(x_1, \cdots, x_n) = \begin{cases} y & \text{if } f(y, x_1, \ldots,) = 0 \text{ and if} \\ & \forall i < y, f(i, x_1, \ldots, x_n) \neq 0 \\ diverges & \text{otherwise} \end{cases}$$

The minimization of f is denoted by:

$$h(x_1, \cdots, x_n) = \mu y(f(y, x_1, \cdots, x_n))$$

The function $f : \mathbb{N}^n \to \mathbb{N}$ is called partial recursive if f is an initial function or if it is obtained by the usage of composition, primitive recursion and minimization from the initial functions. If f is also a total function, it is simply stated that f is a recursive function.

V. The Gödel β Functions

The programming languages include certain programming facilities in order to manipulate vectors, matrixes, etc. Likewise, the Gödel β functions are introduced as an mathematical utility for manipulating tuples. This functions transform tuples into natural numbers, and are also known as pairing functions. Shall define the primitive recursive functions $\beta : \mathbb{N}^2 \to \mathbb{N}$, ${}_1\beta_2 : \mathbb{N} \to \mathbb{N}$ and ${}_2\beta_2 : \mathbb{N} \to \mathbb{N}$ by:

$$\begin{array}{rcl} \beta(x, y) & = & x + \frac{(x+y).(x+y+1)}{2} \\ {}_1\beta_2(\beta(x, y)) & = & x \\ {}_2\beta_2(\beta(x, y)) & = & y \end{array}$$

The β function provides a linearization of the ordered pair according to sequence of Fig. 2 such as it is performed by space filling curves. Besides, the functions ${}_1\beta_2$ and ${}_2\beta_2$ implement the inverse operation of this linearization.

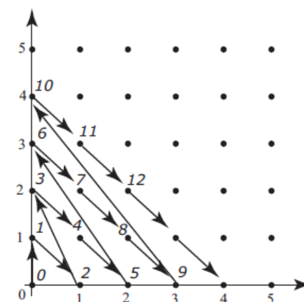

Figure 2. Gödel β pairing function.

Let $\beta_r : \mathbb{N}^r \to \mathbb{N}$ be inductivelly defined by:

$$\begin{array}{rcl}\beta_1(x) & = & x \\ \beta_2(x_1, x_2) & = & \beta(x_1, x_2) \\ \beta_{k+1}(x_1, \ldots, x_{k+1}) & = & \beta(\beta_k(x_1, \ldots, x_k), x_{k+1}),\ k \geq 2 \\ \bar{\beta}(\underline{x}_m, \underline{y}_n) & = & < x_1, \ldots, x_m, y_1, \ldots, y_n > \end{array}$$

VI. Abstract Family of Algorithms (AFA)

Each programming language is conceived to be applied to a specific problem type, and for this reason, those programming languages provide several features oriented to their purpose. However, the Church-Turing Thesis states that all languages are equivalent, and as a consequence, any kind of programming language may reach a problem solution whatever is the problem nature since this solution exists. In fact, an algorithm may be implemented by several programming languages, and as a result, all tasks resolved by the algorithm must also be resolved by its implementations [2]. Hence, the programming languages are materializations of algorithms, responsible for computing functions.

An algorithm is a set of deterministic procedures which are applied to a symbolic input class that eventually may result, for each input, a symbolic output [1]. Note that an algorithm is always finite even though it execution not necessary is finite. This happens because the algorithm is a symbol string of an alphabet Σ.

The set of all possible symbols combination, regardless the generated string size, is known as Σ^*. So, this combination produces non-meaning string, but it also produces the family of all algorithms A describable by using alphabet Σ. Fig. 3 highlight that for each algorithm there is a natural number m associated by a biunivocal relationship ε_A. The relationship ε_A is a primitive recursive function that enumerates all the representable algorithms using alphabet Σ, and so each algorithm is indexed by a $m \in \mathbb{N}$.

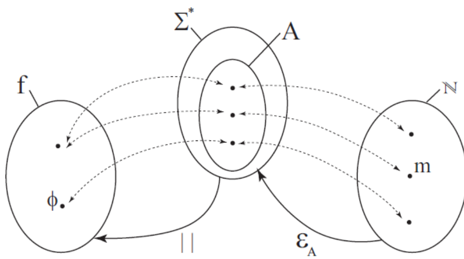

Figure 3. Relationships associated to an Abstract Family os Algorithms.

In addition, each algorithm A has a semantic meaning that is the consequence of its execution, denoted by the operator $|\ |$. It is the computation ϕ of an algorithm $A_m \in A$, and ultimately, it is the computation related to index m of an enumeration because $\phi_m = |A_m| = |\varepsilon_A(m)|$.

An AFA, described by a language L with a finite alphabet Σ, is an enumerable set where each algorithm is associated to a natural number, its index, and a partial function. Given an algorithm of this AFA, it is possible to obtain its index, and given a natural number, it is possible to obtain the algorithm associated to this index. For a given enumerable infinite alphabet, the enumeration must bem more sophisticated, such as the Gödel enumeration [5], based on prime numbers. However, considering the programming languages, the finite alphabet constrain is always applicable.

In spite of the biunivoque relationship ε_A, the relationship $|\ |$ allows the association between one element of f with more than one element of A, which means that may exist more than one algorithm capable to perform a unique task. This set f is composed by partial recursive functions [3][4], that is, functions built by using the initial functions, primitive recursive functions, primitive recursion, minimization and repetition.

***Definition** 2 (*Abstract Family of Algorithms*):* An AFA is a triple $\phi =< A, \varepsilon_A, |\ | >$ where:

- A: is an enumerable set of objects called algorithms.
- ε_A: is an function of $\mathbb{N}$ in A called ϕ enumeration, where $\varepsilon_A(\mathbb{N}) = A$.
- $|\ |$: is an function of A in $\mathbb{N}^{\mathbb{N}}$ called computation.

with the following properties:

- P_1 For all partial recursive functions ϕ, $f : \mathbb{N} \to \mathbb{N}$ exits $m \in \mathbb{N}$ where $\phi_m = |\varepsilon_A(m)|$. When it is applied to an string x, it is written $\phi_m(x)$;
- P_2 Exists a natural number $u_{\mathcal{F}}$ which indexes $\phi_{u_{\mathcal{F}}} = |\varepsilon_A(u)|$ where if x is an input data string of an algorithm, then $\phi_{u_{\mathcal{F}}}(n, x) = \phi_n(x)$;
- P_3 Exists a natural number $c_{\mathcal{F}}$ which indexes $\phi_{c_{\mathcal{F}}} = |\varepsilon_A(c_{\mathcal{F}})|$ where if n and m are an input data string of an algorithm, then $\phi_{c_{\mathcal{F}}(n,m)} = \phi_n \circ \phi_m$. When it is applied to a string x, it is written $\phi_{c_{\mathcal{F}}(n,m)}(x) = \phi_n \circ \phi_m(x)$;

The property P_1 indicates that all partial recursive functions are computed by an algorithm from the AFA $\mathcal{F}$ indexed by a natural number m. The property P_2 is important because it concerns to the universal function of $\mathcal{F}$ whose execution is the appliance of the algorithm of index n for the input data x. Note that the function $\phi_{u_{\mathcal{F}}}$ is associated to an algorithm that executes algorithms, a meta-algorithm, sometimes called universal algorithm. Despite this concept is special, the meta-algorithms are not rare but quite common, such as compilers and operating systems. Finally, the composition property P_3 aims to obtain, from two programs ϕ_n and ϕ_m, a new program $\phi_{n \circ m}$, and to accomplish this task it is necessary that the function $\phi_{c_{\mathcal{F}}}$ provides an modification on ϕ_n and ϕ_m in such way to avoid conflicts between the names of the variables and others tags. Thus, it isolates the algorithm scopes, forbidding their overlap.

The following theorem is known as the Parameterization Theorem or the s-m-n Theorem [1][11][9], and it has a great importance recursive functions theory. It reflects the possibility of packing function arguments in order to obtain programs that use the mechanism of argument passing to increase their modularity and reduce their coupling.

Let a function ϕ with $m + n$ arguments. Suppose that the first m arguments are fixed and the other n arguments allowed to change, resulting into a ϕ function with n arguments. The index of this functions depends on the index of the original function ϕ and the m arguments $x_1, \ldots, x_m$.

***Theorem** 1 (s-m-n Theorem):* For any acceptable $\phi_0, \phi_1, \ldots$, there is a total recursive function $s : \mathbb{N}^{m+1} \to \mathbb{N}$, where for all values $e, x_1, \ldots, x_m, y_1, \ldots, y_n$, with $m, n \geq 1$, we have:

$$\phi_{s(e,x_1,\ldots,x_m)}(y_1, \ldots, y_n) = \phi_e(x_1, \ldots, x_m, y_1, \ldots, y_n)$$

By taking $m = n = 1$, it is obtained the s-m-n theorem in its simple form,

$$\phi_{s(e,x)}(y) = \phi_e(x, y)$$

In order to demonstrate this theorem, let $j, k \in \mathbb{N}$, such as $j, k \geq 1$, f and g primitive recursive functions defined as:

$$\begin{array}{ll} (i) & f(k) = \bar{\beta}(0, k) \\ (ii) & g(\bar{\beta}(j, k)) = \bar{\beta}(j + 1, k) \end{array}$$

As $\mathcal{F}$ is an AFA, there are indexes p and q for f and g, where $f = \phi_p^{\mathcal{F}}$ and $g = \phi_q^{\mathcal{F}}$. Let H be defined by:

$$\begin{array}{lcl} H(0) & = & p \\ H(x + 1) & = & q \circ H(x) \end{array}$$

and so H is primitive recursive. First, it is necessary to demonstrate by induction that $\phi_{H(j)}(k) = \bar{\beta}(j, k)$.

- For $j = 0$

$$\phi_{H(0)}(k) = \phi_p(k) = f(k) = \bar{\beta}(0, k)$$

- Now, suppose that the proposition is valid for j, that is:

$$\phi_{H(j)}(k) = \bar{\beta}(j, k)$$

So, we have:

$$\begin{array}{lcl} \phi_{H(j+1)}(k) & = & \phi_{q \circ H(j)}(k) \\ & = & \phi_q \circ \phi_{H(j)}(k) \\ & = & g(\phi_{H(j)}(k)) \\ & = & g(\bar{\beta}(j, k)) \\ & = & \bar{\beta}(j + 1, k) \end{array}$$

Let k be the index of the function $\phi_k(\bar{\beta}(\bar{\beta}(\underline{x}_m), \bar{\beta}(\underline{y}_n))) = \bar{\beta}_{m+n}(\underline{x}_m, \underline{y}_n)$. We define $s(e, \bar{\beta}(\underline{x}_m)) = e \circ k \circ H(\bar{\beta}(\underline{x}_m))$, so:

$$\begin{array}{l} \phi_{s(e,\bar{\beta}(\underline{x}_m))}(\bar{\beta}(\underline{y}_n)) = \\ = \phi_e \circ \phi_k \circ \phi_{H(\bar{\beta}(\underline{x}_m))}(\bar{\beta}(\underline{y}_n)) \\ = \phi_e \circ \phi_k(\bar{\beta}(\bar{\beta}(\underline{x}_m), \bar{\beta}(\underline{y}_n))) \\ = \phi_e(\bar{\beta}_{m+n}(\underline{x}_m, \underline{y}_n)) \\ = \phi_e(\underline{x}_m, \underline{y}_n) \end{array}$$

This theorem claims that for a given program with $m + n$ input arguments, if m arguments are set to be fixed, than it is possible to get a specialized new program with n input arguments. For example, suppose an application designed to attack cryptographic systems configured by $m + n$ variables. In this application, m arguments describe the cryptographic protocol features such as key Exchange, authentication, signature, etc; and n arguments describe cryptographic techniques such as key length, key manager, algorithm type, etc. The first parameters specify the cryptographic agreement between the actors of the process, and the parameters specify the group of methods used to construct the cryptographic object. If the application designed to attack cryptographic systems is sold for an entity concerned to messages that use public-key authentication cryptography, than the s-m-n Theorem ensures that there is an specific instance of this application that simulates the behavior of an attack against the public-key authentication cryptography by simply instancing the correspondent m parameters.

Another important theorem from the AFA is called Translation Theorem. It provides a relationship between two different AFA.

***Theorem** 2 (Weak Translation Theorem):* Given two abstract algorithm families $\mathcal{F}$ e $\mathcal{G}$ there is a primitive recursive function $\underline{tr}_{\mathcal{F}}^{\mathcal{G}}$, $\mathcal{G} \to \mathcal{F}$ such that for all $m \in \mathbb{N}$:

$$\phi_m^{\mathcal{G}} = \phi_{\underline{tr}_{\mathcal{F}}^{\mathcal{G}}(m)}^{\mathcal{F}}$$

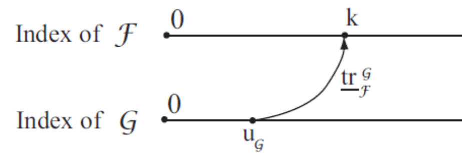

Figure 4. Translation from $\mathcal{G}$ to $\mathcal{F}$ illustrating the Weak Translation Theorem.

The Fig. 4 illustrates that there is a transformation of an indexed algorithm $\mathcal{G}$ whose image is an algorithm indexed on $\mathcal{F}$. In order to demonstrate the existence of this function $\underline{tr}_{\mathcal{F}}^{\mathcal{G}}$, keep in mind that as any AFA, $\mathcal{G}$ has an universal algorithm indexed by $u_{\mathcal{G}}$ which allows to say that $\phi_{u_{\mathcal{G}}}^{\mathcal{G}}(m, x) = \phi_m^{\mathcal{G}}(x)$. Moreover, it is important to remember that the AFA $\mathcal{F}$ computes all computable recursive functions, including the AFA $\mathcal{G}$ universal function. Thus, if k is the index of the function $\phi_{u_{\mathcal{G}}}^{\mathcal{G}}$ in AFA $\mathcal{F}$, it is defined $\underline{tr}_{\mathcal{F}}^{\mathcal{G}}$ by:

$$\underline{tr}_{\mathcal{F}}^{\mathcal{G}}(y) = s(k, y)$$

where y is the index of the algorithm in $\mathcal{G}$ on which to find its equivalent. So:

$$\phi_{\underline{tr}_{\mathcal{F}}^{\mathcal{G}}(m)}^{\mathcal{F}}(x) = \phi_{s(k,m)}^{\mathcal{F}}(x)$$

By using the s-m-n Theorem on $\phi_{s(k,m)}^{\mathcal{F}}(x)$ we have:

$$\begin{array}{lcl} \phi_{\underline{tr}_{\mathcal{F}}^{\mathcal{G}}(m)}^{\mathcal{F}}(x) & = & \phi_{s(k,m)}^{\mathcal{F}}(x) \\ & = & \phi_k^{\mathcal{F}}(m, x) \\ & = & \phi_{u_{\mathcal{G}}}^{\mathcal{G}}(m, x) \\ & = & \phi_m^{\mathcal{G}}(x) \end{array}$$

The Weak Translation Theorem permits to associate one AFA to another. It seems to be natural that this theorem will be used at the next sections to qualify the relationship between algorithms from two different abstract algorithm families. Furthermore, it will be important to support some considerations about Church-Turing Thesis.

One of the most important results for an Abstract Algorithm Family is called the Recursion Theorem [1][6]. This theorem will be presented in the next section.

VII. Recursion Theorem

Let $\mathcal{F} =< A, \varepsilon_A, |\,| >$ be an AFA and $f : \mathbb{N} \to \mathbb{N}$ a total recursive function which transforms a given algorithm. The f specification states that when it is applied to an algorithm index it results into a new index, and there is no need to consider what is the new indexed algorithm. Suppose that the function f applied to an index i maps a new index $f(i)$ associated to an algorithm $\varepsilon_A(f(i))$. A specific relevant scenario occurs when an ordinary algorithm $\varepsilon_A(j)$ computes exactly the same partial recursive function ϕ that the algorithm $\varepsilon_A(f(i))$ does, that is, $|\varepsilon_A(f(i))| = |\varepsilon_A(j)|$. In fact, it is expected that there will be algorithm capable of executing the same task, and also, there might be an unlimited relationships of this kind.

Hence, the set of all algorithms that compute the same partial recursive function ϕ is given by:

$$\mathcal{T}_{\mathcal{F}} = \left\{ < i, j > \in \mathbb{N}^2 | \phi_{f(i)} = \phi_j \right\}$$

***Theorem** 3 (Fixed Point Theorem):* The particular case where the algorithm $\varepsilon_A(f(i))$ computation produces the same function produced by $\varepsilon_A(i)$ is a circumstance when it is said that $i \in \mathbb{N}$ is a fixed point of ϕ [1][4], that is,

$$\phi_{f(i)} = \phi_i$$

A function fixed point is a value which remains unchanged, even though the function is applied to it. Note that the self-reference usage tends to be undesirable by the mathematical logic, such as in the lair paradox. However, the self-reference is important since the primitive recursion is based on it, that is, it is based on function definition in terms of the same functions.

The Recursion Theorem, as will be demonstrated in the following, is related to mathematical logic and self-reproducing systems, that is, functions that produce functions, and finally, the theorem is correlated to the possibility of creating machines that constructs replicas of them. Usually, non-researches of this subject have a huge resistance to accept this possibility, mainly because they accept as a dogma that machines cannot self-reproduce. This is clearly a mistake and the reason will be explained.

The mistaken reasoning works as follows: first, consider the matter of a machine be capable to produce another machine, such as a microprocessors factory. Electronic supplies are provided to this factory, it employs a robotic manufactory according a pre-defined instruction set, and at the end of the process it deploys a microprocessor. This factory must be more complex than the produced microprocessors since this factory needs to internalize not only the microprocessor design, but also the governance project of all its robots. The same reasoning might be used for a more abstract situation, where a machine A constructs a machine B. Therefore, the machine A must be more complex than the machine B. Additionally, it is correct to assert that any machine cannot be more complex than itself. Consequently, no machine can construct itself, turning the self-reproduction an impossible operation. Nevertheless, the Recursion Theorem refutes categorically this conclusion. The main explanation is because it is wrong the argument that there is a relationship between the ability of executing several functionalities and the possibility of replicating these functionalities.

***Theorem** 4 (Recursion Theorem):* Given an AFA and a recursive function f, there is a natural number n where

$$\phi_{f(n)} = \phi_n$$

This theorem means that there is a natural number n associated to a partial recursive function ϕ which submitted to a function f produces as a result the same function ϕ. Therefore, this procedure replicates the function ϕ.

In order to demonstrate this property, let n be a natural number and A_n an algorithm dependant on n with the following specification:

1) Takes n as input and applies ε_A to it;
2) Selects the algorithm $\varepsilon_A(n)$ from the AFA $\mathcal{F}$ to be used;
3) Computes $\varepsilon_A(n)$ using the same entrance n, and so obtainning $|\varepsilon_A(n)|\,(n)$;

$$A_n = \begin{cases} |\varepsilon_A(|\varepsilon_A(n)|\,(n))| & se\ \ |\varepsilon_A(n)|\,(n)\ \ converge \\ \\ diverge & se\ \ |\varepsilon_A(n)|\,(n)\ \ diverge \end{cases}$$

The classical Mathematics that ordinary function, when submitted to an input, will always produce an output, no matter what is this result. On the other hand, the Computability does not share this security degrees with classical Mathematics. An algorithm may receive a certain input and it may never stop processing it, an issue known as the halting problem. Under this point of view, the A_n specification needs to consider the divergent situation into its definition.

The Fig. 5 shows the strategy A_n, as previously defined, that will be used in order to demonstrate the theorem. The A_n goal is to show that n is a fixed point of ϕ_n. To accomplish this, it is necessary to choose the ϕ associated to n, compute the result of $\phi_n(n)$, and then, it is essential to demonstrate that this result remains unchanged when it is submitted to ϕ_n again.

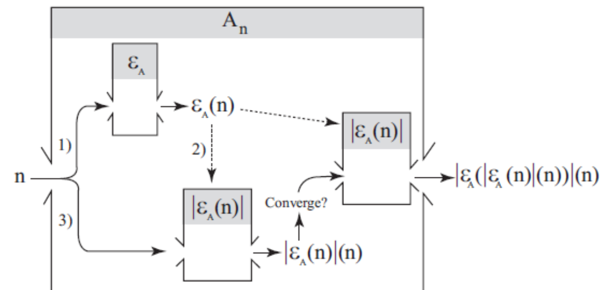

Figure 5. Recursion Theorem demonstrating strategy using the proposed algorithm A_n.

The property P_2 of the AFA $\mathcal{F}$ allows to say that:

$$|\varepsilon_A(n)|\,(n) = \phi_{u_{\mathcal{F}}}(n, n)$$

which indicates that the meta-algorithm will provide the execution of the algorithm indexed by n and whose input will be the string n. So, let n be a natural number submitted to the execution stream of Fig. 5. Therefore:

$$\begin{aligned} |A_n|(y) &= |\varepsilon_A(|\varepsilon_A(n)|(n))|(y) \\ &= \phi_{u_\mathcal{F}}(|\varepsilon_A(n)|(n), y) \\ &= \phi_{u_\mathcal{F}}(\phi_u(n,n), y) \end{aligned}$$

Let H be a function defined by:

$$H(n,y) = |A_n|(y) = \phi_{u_\mathcal{F}}(\phi_{u_\mathcal{F}}(n,n), y)$$

and let m_H the index of H. Define g by:

$$g(n) = s(m_H, n)$$

Then, performing the g substitution and by using s-m-n Theorem in its simple form we have:

$$\phi_{g(n)}(y) = \phi_{s(m_H,n)}(y) = \phi_{m_H}(n,y) = H(n,y)$$

Let $n_f = g(c_\mathcal{F}(m_f, m_g))$ an ordinary natural number which specifies a certain algorithm $\varepsilon_A(n_f)$, and c a composition function that merges the machine index m_f, proposed on the theorem, and the machine m_g. Computing this algorithm $\varepsilon_A(n_f)$ for the entry y we obtain:

$$\begin{aligned} \phi_{n_f}(y) &= \phi_{g(c_\mathcal{F}(m_f,m_g))}(y) \\ &= H(c_\mathcal{F}(m_f,m_g), y) \\ &= \phi_{u_\mathcal{F}}(\phi_{u_\mathcal{F}}(c_\mathcal{F}(m_f,m_g), c_\mathcal{F}(m_f,m_g)), y) \\ &= \phi_{u_\mathcal{F}}(\phi_{c_\mathcal{F}(m_f,m_g)}(c_\mathcal{F}(m_f,m_g)), y) \\ &= \phi_{c_\mathcal{F}(m_f,m_g)(c_\mathcal{F}(m_f,m_g))}(y) \\ &= \phi_{f \circ g(c_\mathcal{F}(m_f,m_g))}(y) \\ &= \phi_{f(g(c_\mathcal{F}(m_f,m_g)))}(y) \\ &= \phi_{f(n_f)}(y) \end{aligned}$$

So $n_f = g(c_\mathcal{F}(m_f, m_g)) = s_F(m_H, c_\mathcal{F}(m_f, m_g))$ is the fixed point of ϕ.

The Recursion Theorem can be used to construct some interesting recursive functions, such as the Self-replication Theorem [15].

***Theorem** 5 (Self-replication Theorem):* There is an algorithm that prints its own description, given any input.

Define f by $\phi_{f(x)}(y) = x$, that is, $f(x) = s(e,x)$ where e is an index for the projection function so that $\phi_{f(x)}(y) = \phi_{s(e,x)}(y) = \phi_e(x,y) = \underline{pr}^2_1(x,y) = x$. Since $f(x)$ is recursive, by the Recursion Theorem there is a n such that $\phi_{f(n)} = \phi_n$, hence $\phi_n(y) = \phi_{f(n)}(y) = n$.

The function ϕ_n is a function whose constant value is its index n. The algorithm which computes ϕ_n always outputs its own description. The word n is called a description of the algorithm because the algorithm with index n is $\varepsilon_A(n)$.

As a result, there is a natural number n associated to an AFA $\mathcal{F}$ that once submitted to function f it produces, as a result, the function ϕ. A computer virus and a computer worm [8] are designed to replicate themselves among several computers. These viruses are inactive when analyzed exclusively as block of programming code. However, when deployed into a host, it may become active and may start to transmit copies of itself to other accessible computers. Consequently, with the purpose of accomplishing its replicating task, the viruses contain the type of scheme described at the Recursion Theorem demonstration [10]. The quines are another variety of this kind of application [9]. They are programs with no entry that produce copies of themselves.

VIII. Isomorphism Theorem

In this section we will present some consequences of the: Fixed Point Theorem, Translation Theorem and Recursion Theorem. These issues will provide support to the main topic called Isomorphism Theorem.

Let $x_1 \neq x_2$ such that $\phi^\mathcal{G}_{x_1}(y) \neq \phi^\mathcal{G}_{x_2}(y)$. With the assistance of the universal algorithm from the AFA $\mathcal{G}$, it is possible to say that $\phi^\mathcal{G}_{u_\mathcal{G}}(x_1, y) \neq \phi^\mathcal{G}_{u_\mathcal{G}}(x_2, y)$. Though, the universal algorithm from $\mathcal{G}$ has an index k on AFA $\mathcal{F}$. So:

$$\begin{aligned} \phi^\mathcal{G}_{u_\mathcal{G}}(x_1,y) &\neq \phi^\mathcal{G}_{u_\mathcal{G}}(x_1,y) \\ \phi^\mathcal{F}_{k}(x_1,y) &\neq \phi^\mathcal{F}_{k}(x_1,y) \\ \phi^\mathcal{F}_{s(k,x_1)}(y) &\neq \phi^\mathcal{F}_{s(k,x_2)}(y) \\ \phi^\mathcal{F}_{\underline{tr}^\mathcal{G}_\mathcal{F}(x_1)}(y) &\neq \phi^\mathcal{F}_{\underline{tr}^\mathcal{G}_\mathcal{F}(x_2)}(y) \end{aligned}$$

If, for the same entry y two algorithm produce distinct computations, then it is because these algorithm are also distinct, and consequently are their indexes. Therefore, $\underline{tr}^\mathcal{G}_\mathcal{F}(x_1) \neq \underline{tr}^\mathcal{G}_\mathcal{F}(x_2)$. If this happens and if $x_1 \neq x_2$ then the function $\underline{tr}^\mathcal{G}_\mathcal{F}$ must be injective. For this reason, an injective $\underline{tri}^\mathcal{G}_\mathcal{F} = \underline{tr}^\mathcal{G}_\mathcal{F}$.

***Theorem** 6 (Strong Translation Theorem):* Given two abstract algorithm families $\mathcal{F}$ e $\mathcal{G}$ there is a primitive recursive injective function $\underline{tri}^\mathcal{G}_\mathcal{F}$, $\mathcal{G} \to \mathcal{F}$ such that for all $m \in \mathbb{N}$:

$$\phi^\mathcal{G}_m = \phi^\mathcal{F}_{\underline{tri}^\mathcal{G}_\mathcal{F}(m)}$$

Moreover, pay attention to some features from a peculiar function p called padding function. Its role is to add instructions to the algorithm without changing its functionality. At first, it may seems to be useless under the traditional computing point of view, but this behavior occur in several everyday forms. For instance, when a programmer add coments to his programming code, the size of the text increases, and the desired execution behavior is kept unchanged. So, the act of documenting the source-code of a program is a padding expression.

The current programming languages clearly prioritize the usage of code interpretation (Java, PHP, IL from .Net, script languages, ...), against code compilation. The main reason is the benefits provided by the code portability. Though, the side effect is that the interpreted code may easily be decompilated. Notwithstanding, software developers are still building interpreted application, distributing them through download or physical medias. Under these circumstances, it is critical to protect application source-code and intellectual capital, no matter if this application is freeware, shareware, or commercially licensed. The traditional cryptography is not a solution for this problem, because it would prevent the computer processor to have access to the program instructions. Therefore, the cryptographic solution is the code obfuscation, that is, to make the source-code less understandable for a human by adding irrelevant and outwit instructions, but keeping the functionality, exactly as the code padding.

The understanding of how the padding operates is better comprehended by using Labeled Markov Algorithm language [4],which will be summarized as follows.

***Definition** 3 (Labeled Markov Algorithm):* A LMA with an input Σ, or shortly a LMA in Σ, is a sequence of $n > 0$ expressions such as $l:\ x \to\ y/l'$, called commands, where:

- l and l' are natural numbers, or their representation, known as command labels and transferring labels, respectively.
- x is a word from the alphabet, called command' left.
- y is a word from the alphabet, called command' right.

Additionally, for all $i,\ 1 \leq i \leq n$, the i command has the $i-1$ label.

As it is common on programming languages, the sequence commands are separated by the symbol ;. Despite the data do not belong to the LMA definition (there are no variables or registers), the central point is to submit a Σ^* string to the LMA. The input string is transformed by the execution of the LMA, and its output is a modified string. Let a LMA be $\varepsilon_A(e)$ and $l : x_l \rightarrow y_l/l'$ a command from $\varepsilon_A(e)$. It is defined the function $\phi_e = |l : x_l \rightarrow y_l/l'|$ with values given by:

$$|l : x \rightarrow y/l'|(w) = \begin{cases} \underline{subst}(w, x, y) & if\ x \preceq w \\ \\ w & if\ x \not\preceq w \end{cases}$$

where $\underline{subst}(w, x, y)$ is the substitution result of the first occurrence of x in w for y, and $x \preceq w$ means that x is a substring from w.

The following sequence of commands is a LMA form from an alphabet Σ:

$$\begin{array}{l} 0 : 0 \rightarrow a/1; \\ 1 : a1 \rightarrow 1a/1; \\ 2 : a2 \rightarrow 2a/1; \\ 3 : a \rightarrow 2/5; \end{array}$$

Let $w \in \Sigma^*$. The command label 0 means: substitute the first occurrence of the null word in w for a, that is, put the marker a at the beginning of the word w and go to the execution of command label 1. The command label 1 means: substitute the first occurrence of the word $a1$ in w for $1a$ and execute again the command label 1.

Intuitively, it is known that an algorithm may be implemented into several different ways, even though theses different implementations compute the same function. As the index i from a function φ_i is given by the LMA enumeration $\varepsilon_A(i)$ that compute this function, in order to ensure the mapping to infinite indexes, it sufficient to take $\varepsilon_A(i)$ and add innocuous commands to produce a new $\varepsilon_A(i')$.

Therefore, let $p(e, x)$ be a padding function, recusive primitive and injective, which concatenates the algorithm $\varepsilon_A(e)$ with an irrelevant algorithm $\varepsilon_A(x)$. The concatenation operation may occur at the beginning ou at the end of the algorithm, but it may also more sophisticated and intercalate patches of algorithm $\varepsilon_A(e)$ with patches from $\varepsilon_A(x)$. Whatever the case, we have $\phi_{p(e,x)} = \phi_e$ and if $p(e, x) = p(e, y)$ then it is because $x = y$.

The main idea is to concatenate the useless commands from algorithm $\varepsilon_A(x)$ with the LMA algorithm index$e$. So, if $\varepsilon_A(e)$ is the LMA:

$$\varepsilon_A(e) = \begin{cases} 0 : x_0 \rightarrow y_0/l_0; \\ 1 : x_1 \rightarrow y_1/l_1; \\ \ldots \\ k : x_k \rightarrow y_k/l_k; \end{cases}$$

Let x be the index of the program:

$$\varepsilon_A(x) = \begin{cases} 0 : 0 \rightarrow 0/1; \\ 1 : 0 \rightarrow 0/2; \\ \ldots \\ y : 0 \rightarrow 0/y+1; \end{cases}$$

The LMA $\varepsilon_A(p(e, x))$ is the LMA:

$$\varepsilon_A(p(e, x)) \begin{cases} 0 : 0 \rightarrow 0/1; \\ 1 : 0 \rightarrow 0/2; \\ \ldots \\ y : 0 \rightarrow 0/y+1; \\ y+1 : x_0 \rightarrow y_0/l_0; \\ y+2 : x_1 \rightarrow y_1/l_1; \\ \ldots \\ y+k+1 : x_k \rightarrow y_k/l_k; \end{cases}$$

Then, the program created by the concatenation is the composition of the programs indexes x e e, and so it is sufficient to take $p(e, x) = c_{\mathcal{F}}(e, x)$, because then:

$$\phi_{p(e,x)} = \phi_{c_{\mathcal{F}}(e,x)} = \phi_e$$

Remembering that by definition the function p is injective.

Let $\mathcal{F}$ and $\mathcal{G}$ be two abstract algorithm families. Then, there is a recursive function $\underline{trias}_{\mathcal{G}}^{\mathcal{F}}$ such that for all $x \in \mathbb{N}$:

$$\phi^{\mathcal{G}}_{\underline{tris}^{\mathcal{G}}_{\mathcal{F}}(x)} = \phi^{\mathcal{F}}_x$$

and additionally:

$$0 < \underline{tris}^{\mathcal{G}}_{\mathcal{F}}(x) < \underline{tris}^{\mathcal{G}}_{\mathcal{F}}(x+1)$$

In order to make this happens, it is necessary to translate each program x and then add superfluous commands until the desired length is obtained. Note that this feature from function $\underline{tris}^{\mathcal{G}}_{\mathcal{F}}$ allows to state that for all algorithm from A, there is a natural number $x \in \mathbb{N}$ such that this algorithm is determined by $\varepsilon_A(x)$, characterizing a surjection.

Finally, it is possible to present the Isomorphism Theorem [1][4], which ensures a bijective mapping between abstarct algorithm families, that is, all abstract algorithm families are equivalent. The Isomorphism Theorem shows not only two acceptable algorithms might be ttranslated from one to another, but also that there is a one-to-one translating between these two acceptable algorithms.

***Theorem** 7 (Isomorphism Theorem):* Let $\mathcal{F}$ and $\mathcal{G}$ be two abstract algorithm families. If the translation fuction is injective and surjective then there is a bijective recursive function $\underline{trb}^{\mathcal{G}}_{\mathcal{F}}$ such that for all $x \in \mathbb{N}$:

$$\phi^{\mathcal{G}}_{\underline{trb}^{\mathcal{G}}_{\mathcal{F}}(x)} = \phi^{\mathcal{F}}_x$$

The demonstration os this theorem involves the usage of the function $\underline{tris}$, and it is presented a more complete adaptation from the one described by Machtey e Young [16]. As it was studied, this function is injective, that is, distinct indexes i and j from $\mathcal{G}$, are mapped through translation into distinct indexes from $\mathcal{F}$. This function is also surjective because all indexes from $\mathcal{F}$ are translation images from a index in $\mathcal{G}$.

Note that $\underline{tris}^{\mathcal{G}}_{\mathcal{F}}$, by construction, is monotonically increasing. Thus, given a natural number x, the composition result

$\underline{tris}_{\mathcal{G}}^{\mathcal{F}} \circ \underline{tris}_{\mathcal{F}}^{\mathcal{G}}$ produces a numerical value greater than x, that is, $\underline{tris}_{\mathcal{G}}^{\mathcal{F}}(\underline{tris}_{\mathcal{F}}^{\mathcal{G}}(x)) > x$. Once the function $\underline{tris}$ is based on padding we have that $x < \underline{tris}_{\mathcal{F}}^{\mathcal{G}}(x) < \underline{tris}_{\mathcal{G}}^{\mathcal{F}}(\underline{tris}_{\mathcal{F}}^{\mathcal{G}}(x))$. Consequently, the inverse translation functions must be monotonically decreasing. Therefore, we define the inverse functions s^* e t^* by (see Fig. 6):

$$s^*(x) = \begin{cases} y & \text{if exists } y \text{ such that } \underline{tris}_{\mathcal{F}}^{\mathcal{G}}(y) = x \\ 0 & \text{otherwise} \end{cases}$$

$$t^*(x) = \begin{cases} y & \text{if exists } y \text{ such that } \underline{tris}_{\mathcal{G}}^{\mathcal{F}}(y) = x \\ 0 & \text{otherwise} \end{cases}$$

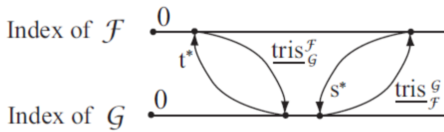

Figure 6. The inverse functions s^* e t^* associated to the translations between $\mathcal{F}$ and $\mathcal{G}$.

Given an index x from AFA $\mathcal{F}$, it is possible to define a sequence:

$$I_x = \{x, s^*(x), t^* \circ s^*(x), s^* \circ t^* \circ s^*(x) \cdots\}$$

As s^* and t^* are strictly decreasing (see Fig. 7), this sequence has, at most, $x+1$ elements, and also, the last value is always 0, that is, for some $l \in \mathbb{N}$:

1) the sequence ends on $\mathcal{F}$, and then $(t^* \circ s^*)^l(x) = 0$, or
2) the sequence ends on $\mathcal{G}$, and then $s^* \circ (t^* \circ s^*)^l(x) = 0$.

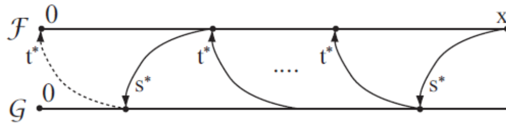

Figure 7. The s^*'s and t^*'s strictly decreasing sequences converging to 0 as the last value.

The dot line from Fig. 7 indicates that the sequence ends in $\mathcal{F}$ according to one of the options:

- $\underline{tris}_{\mathcal{G}}^{\mathcal{F}}(0) = (t^* \circ s^*)^l(x)$, or;
- $\nexists y$ in $\mathcal{F}$ such that $\underline{tris}_{\mathcal{G}}^{\mathcal{F}}(y) = (t^* \circ s^*)^l(x)$.

Analogously, note that a sequence ends on $\mathcal{G}$ according two situations:

- $\underline{tris}_{\mathcal{F}}^{\mathcal{G}}(0) = s^* \circ (t^* \circ s^*)^{l-1}(x)$, or;
- $\nexists y$ in $\mathcal{G}$ such that $\underline{tris}_{\mathcal{G}}^{\mathcal{F}}(y) = s^* \circ (t^* \circ s^*)^{l-1}(x)$.

Lets define the function:

$$\underline{term\mathcal{G}}(x) = \begin{cases} 1 & if\ I_x\ ends\ on\ \mathcal{G} \\ 0 & if\ I_x\ ends\ on\ \mathcal{F} \end{cases}$$

And the function:

$$\underline{trb}_{\mathcal{F}}^{\mathcal{G}}(x) = \begin{cases} \underline{tris}_{\mathcal{F}}^{\mathcal{G}}(x) & if\ \underline{term\mathcal{G}}(x) = 1 \\ t^*(x) & if\ \underline{term\mathcal{G}}(x) = 0 \end{cases}$$

Hence, it is expected to prove that $\underline{trb}_{\mathcal{F}}^{\mathcal{G}}$ is the desired translation function. Let x be an index from $\mathcal{G}$:

1) if $\underline{term\mathcal{G}}(x)=1$ then $\phi^{\mathcal{G}}_{\underline{trb}_{\mathcal{F}}^{\mathcal{G}}(x)} = \phi^{\mathcal{G}}_{\underline{tris}_{\mathcal{F}}^{\mathcal{G}}(x)} = \phi^{\mathcal{F}}_x$
2) if $\underline{term\mathcal{G}}(x)=0$ then $\phi^{\mathcal{G}}_{\underline{trb}_{\mathcal{F}}^{\mathcal{G}}(x)} = \phi^{\mathcal{G}}_{t^*(x)} = \phi^{\mathcal{F}}_{\underline{tris}_{\mathcal{G}}^{\mathcal{F}}(t^*(x))} = \phi^{\mathcal{F}}_x$

In both cases $\phi^{\mathcal{G}}_{\underline{trb}_{\mathcal{F}}^{\mathcal{G}}(x)} = \phi^{\mathcal{F}}_x$. Finally, it is necessary to prove that $\underline{trb}_{\mathcal{F}}^{\mathcal{G}}$ is bijective.

1) $\underline{trb}_{\mathcal{F}}^{\mathcal{G}}$ is injective: let $x, y \in \mathbb{N}$ such that $\underline{trb}_{\mathcal{F}}^{\mathcal{G}}(x) = \underline{trb}_{\mathcal{F}}^{\mathcal{G}}(y)$. Then it is mandatory that $\underline{term\mathcal{G}}(x) = \underline{term\mathcal{G}}(y)$:
   a) If $\underline{term\mathcal{G}}(x) = 1$ then $\underline{tris}_{\mathcal{F}}^{\mathcal{G}}(x) = \underline{tris}_{\mathcal{F}}^{\mathcal{G}}(y)$ thus, $x = y$;
   b) If $\underline{term\mathcal{G}}(x) = 0$ then $s^*(x) = s^*(y)$ thus, $x = y$;
2) $\underline{trb}_{\mathcal{F}}^{\mathcal{G}}$ is surjective: let y be an index from $\mathcal{F}$, then there is an index x from $\mathcal{G}$ such that $\underline{trb}_{\mathcal{F}}^{\mathcal{G}}(x) = y$, because analysing $\underline{tris}_{\mathcal{F}}^{\mathcal{G}}(y)$, it is verified that:
   a) if $\underline{term\mathcal{G}}(\underline{tris}_{\mathcal{F}}^{\mathcal{G}}(y)) = 1$ then $x = s^*(y)$ once that, in this case, $y = \underline{tris}_{\mathcal{G}}^{\mathcal{F}}(x) = \underline{trb}_{\mathcal{G}}^{\mathcal{F}}(x)$;
   b) if $\underline{term\mathcal{G}}(\underline{tris}_{\mathcal{F}}^{\mathcal{G}}(y)) = 0$ then $x = \underline{tris}_{\mathcal{F}}^{\mathcal{G}}(y)$ once that, in this case, $y = t^*(\underline{tris}_{\mathcal{F}}^{\mathcal{G}}(y)) = \underline{trb}_{\mathcal{F}}^{\mathcal{G}}(x)$;

The importance of this theorem is a reinforcement to Church-Turing Thesis, whatever computer model with the three basic properties (P_1, P_2, P_3), even possessing supposedly more powerful properties, is recursively isomorphic to the already known models. This result is important because it provides a convincing argument to whatever particular computing model is choosen. The Isomorphism Theorem states that all abstract algorithm families as equivalent, and that there do exists an effective and bijective translation function between them.

Note that the Cryptography studies the artifices responsible for transforming the information from its plaintext form into a ciphered form, comprehensible by a select group of people. The evaluation of the ciphered message provides the informative content. The results of the cryptographic techniques implementation comprise some kind of computation, such as the operator $|\ |$.

On the other hand, the Linguistics is the study of the language, that is, the schemes used by mankind to comprehend its ideas and feelings (the authors are consciously avoiding the word "communicate" because the Pêcheux [17] concept of imaginary formation). This comprehension is a consequence of carrying out several human abilities that are associated to an operator of semantic meaning, again, just like the operator $|\ |$.

Therefore, it is possible to incorporate these two points of view because both of them study principles and techniques which define a set of functions whose ultimate objective is to perform an evaluation, such as the algorithms.

By an AFA we mean a denumerable set of undefined objects called algorithms, each of which has associated with exactly one partial n-variable function for each positive integer n. Consequently, we claim that that Linguistics and Cryptography are associated to two different abstract families of algorithms. Moreover, the Isomorphism Theorem provides support to assert that the Linguistics and Cryptography set of objects are equivalent. Thus, a computable function, whose results

are acceptable by one study approach, might be successfully used on the other knowledge area. Undoubtedly, an algorithm $\varepsilon_A^{\mathcal{G}}(n)$ from one knowledge area does not need to be the same one from the other area. However, there do exists an analogous algorithm on the other family, $\varepsilon_A^{\mathcal{F}}(n)$, and the mapping between them is performed by the translation function $\underline{trb}_{\mathcal{F}}^{\mathcal{G}}$, even though this translation might be unknown.

Hence, it seems to be reasonable to use computational linguistics and modern information retrieval to perform cryptographic analysis, not only to recover the informative content of a ciphered message, but also to locate features about the cryptographic key and the type of algorithm used for encryption. Alternatively, the Linguistic comparative method and the pragmatic factors study can be enhanced by Cryptography quantitative and qualitative analysis algorithms.

IX. Conclusions

This article has presented several remarks about how recursive functions computability study can be directly associated to Cryptography themes. In order to accomplish this task, the abstract family of algorithms theory was revised producing a different approach from the ones currently available in literature. The Isomorphism Theorem demonstration provides the technical support for a comparison between two AFA, establishing a connection between Cryptology and Linguistics.

Further studies on this matter include, but not restricted to, a research on cryptogram patterns associated to the keys used on AES and RSA. It also seems to be possible to use corpora techniques to group cryptographic texts according to their cipher keys.

References

[1] Hartley Rogers Jr. *Theory of Recursive Functions and Effective Computability*, 1st ed, McGraw-Hill Book Company, 1967.

[2] Roberto Lins de Carvalho. *Máquinas, Programas e Algoritmos*, 2ª Escola de Computação, Campinas, Universidade Estadual de Campinas, 1981.

[3] Yu I. Manin. *A Course in Mathematical Logic*, 1st ed, Graduate Texts in Mathematics 53, Springer-Verlag, 1977.

[4] Claudia Maria Garcia Medeiros de Oliveira and Roberto Lins de Carvalho. *Modelos de Computação e Sistemas Formais*, 11ª Escola de Computação, Rio de Janeiro, Universidade Federal do Rio de Janeiro, 1998.

[5] Elliott Mendelson. *Introduction to Mathematical Logic*, 3rd ed, Cole Mathematics Series, The Wadsworth and Brooks, 1987.

[6] Michael Sipser. *Introduction to The Theory of Computation*, 2rd ed, Course Technology Series, Thomson, 2006.

[7] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory*, Language and Computation. USA, Addison-Wesley Publishing Company, 1979

[8] Adam Young and Moti Yung. *Malicious Cryptography: Exposing Cryptovirology*, John Wiley and Sons Inc., 2004.

[9] Nigel J. Cutland. *Computability: An introduction to recursive function theory*, Cambridge University Press, 1980.

[10] Guillaume Bonfante and Matthieu Kaczmarek and Jean-Yves Marion. *A Classification of Viruses through Recursion Theorems*, CiE 2007, volume 4497 of Lecture Notes in Computer Science, 2007.

[11] Steven Homer and Alan L. Selman. *Computability and Complexity Theory*, Texts in Computer Science, 2nd ed, Springer, 2011.

[12] Alan M. Turing. *The Undecidable*, chapter On Computable Numbers with an Application to the Entscheidungsproblem, pages 115-151. Raven Press, New York, 1965.

[13] R. J. Nelson. *Introduction to Automata*, John Wiley and Sons, Inc., USA, 1965.

[14] Frederick C. Hennie. *Introduction to Computability*, Series in Computer Science and Information Processing, Addison-Wesley, Reading, Massachusets, USA, 1977.

[15] Walter S. Brainerd and Lawrence H. Landweber. *Theory of Computation*, John Wiley and Sons, 1974.

[16] Michael Machtey and Paul Young. *An Introduction to the General Theory of Algorithms*, Theory of Computation Series, Elsevier North Holland Inc., 1978.

[17] Michel Pêcheux. *Análise Automática do Discurso*, In: Por uma Análise Automática do Discurso, Editors: Françoise Gadet and Tony Hak, Editora Unicamp, 3.ed., 1997.

Roberto Lins de Carvalho did his PhD. on informatics at University of Toronto (1974), MSc. on informatics at Pontifical Catholic University of Rio de Janeiro - PUC Rio (1969), under graduation on telecommunication engineering at Military Engineering Institute - IME (1967) and under graduation military studies at Military Academy of Agulhas Negras - AMAN (1961). Lectured at Pontifical Catholic University of Rio de Janeiro, Campinas University, Fluminense Federal University, Aeronautics Technological Institute and Military Engineering Institute, where supervised several master dissertations and doctoral thesis on theorem proving, knowledge base systems and knowledge representation. Over fourteen years is the leader of Witty Group coaching artificial intelligence computer applications and is retired from Scientific Computing Brazilian National Laboratory - LNCC.

Flávio Luis de Mello did his DSc. on theory of computation and image processing at the Federal University of Rio de Janeiro - UFRJ (2006), MSc. on computer graphics at Federal University of Rio de Janeiro - UFRJ (2003), under graduation on systems engineering at Military Engineering Institute - IME (1998). Developed command and control systems and implemented military messages interchange applications during twelve years as Brazilian Army officer. Responsible for developing software applications based on theorem proving, knowledge base systems and knowledge representation from Witty Group. Associate Professor at the Electronics and Computing Department (DEL) of Polytechnic School (Poli) at Federal University of Rio de Janeiro (UFRJ) since 2007.