The Existence of One-Way Functions

Frank Vega

THE EXISTENCE OF ONE-WAY FUNCTIONS

FRANK VEGA

Abstract. We assume there are one-way functions and obtain a contradiction following a solid argumentation, and therefore, one-way functions do not exist applying the reductio ad absurdum method. Indeed, for every language L that is in EXP and not in P, we show that any configuration, which belongs to the accepting computation of $x \in L$ and is at most polynomially longer or shorter than x, has always a non-polynomial time algorithm that find it from the initial or the acceptance configuration on a deterministic Turing Machine which decides L and has always a string in the acceptance computation that is at most polynomially longer or shorter than the input $x \in L$. Next, we prove the existence of one-way functions contradicts this fact, and thus, they should not exist. Hence, function problems such as the integer factorization of two large primes can be solved efficiently. In this way, this work proves that is not safe many of the encryption and authentication methods such as the public-key cryptography. It could be the case of $P = NP$ or $P \neq NP$, even though there are no one-way functions. However, we prove that $P = UP$.

1. Introduction

The P versus NP problem is the major unsolved problem in computer science. It was introduced in 1971 by Stephen Cook [1]. Today is considered by many scientists as the most important open problem in this field [3]. A solution to this problem will have a great impact in other fields such as mathematics and biology.

During the first half of the twentieth century many investigations were focused on formalizes the knowledge about the algorithms using the theoretical model described by Turing Machines. On this time appeared the first computers and the mathematicians were able to model the capabilities and limitations of such devices appearing precisely what is now known as the science of computational complexity theory.

Since the beginning of computation, many tasks that man could not do, were done by computers, but sometimes some difficult and slow to resolve were not feasible for even the fastest computers. The only way to avoid the delay was to find a possible method that cannot do the exhaustive search that was accompanied by "brute force". Even today, there are problems which have not a known method to solve easily yet.

This property has been used in the security methods inside of practical computational applications using tools such as the suspected one-way functions. If one-way functions do not exist, then this would imply that some algorithms used in cryptography will be easy to break at some point. However, if some functions are one-way, they would ensure that there are hundreds of problems that have not a feasible solution. This is largely derived from this result that $P \neq NP$, so there

2000 *Mathematics Subject Classification.* Primary 68-XX, 68Qxx, 68Q15.

Key words and phrases. Complexity classes, one-way function, P, UP, NP, EXP.

will be a huge amount of problems that can be checked easily but without some practical solution [8]. It will remain the best option to use brute force or a heuristic algorithm in many cases.

We use in this work a method known as reductio ad absurdum which is a common form of argument which seeks to demonstrate that a statement is true by showing that a false or absurd result follows from its denial. This rule has formed the basis in formal fields like logic and mathematics. In this work, we assume there are one-way functions and obtain a contradiction following a solid argumentation, and therefore, one-way functions do not exist applying this method.

2. Theory

The argument made by Alan Turing in the twentieth century proves mathematically that for any computer program we can create an equivalent Turing Machine [9]. A Turing Machine M has a finite set of states K and a finite set of symbols A called the alphabet of M. The set of states has a special state s which is known as the initial state. The alphabet contains special symbols such as the start symbol $\triangleright$ and the blank symbol \$.

The operations of a Turing Machine are based on a transition function δ, which takes the initial state with a string of symbols of the alphabet that is known as the input. Then, it proceeds to reading the symbols on the cells contained in a tape, through a head or cursor. At the same time, the symbols on each step are erased and written by the transition function, and later moved to the left $\longleftarrow$, right $\longrightarrow$ or remain in the same place $-$ for each cell. Finally, this process is interrupted if it halts in a final state: the state of acceptance "*yes*", the rejection "*no*" or halting h [7].

A Turing Machine halts if it reaches a final state. If a Turing Machine M accepts or rejects a string x, then $M(x) =$ "*yes*" or "*no*" is respectively written. If it reaches the halting state h , we write $M(x) = y$, where the string y is considered as the output string, i.e., the string remaining in M when this halts [7].

A transition function δ is also called the "program" of the Turing Machine and is represented as the triple $\delta(q, \sigma) = (p, \rho, D)$. For each current state q and current symbol σ of the alphabet, the Turing Machine will move to the next state p, overwriting the symbol σ by ρ, and moving the cursor in the direction $D \in \{\longleftarrow, \longrightarrow, -\}$ [7]. When there is more than one tape, δ remains deciding the next state, but it can overwrite different symbols and move in different directions over each tape.

Operations by a Turing Machine are defined using a configuration that contains a complete description of the current state of the Machine. A configuration is a triple (q, w, u) where q is the current state and w, u are strings over the alphabet showing the string to the left of the cursor including the scanned symbol and the string to the right of the cursor respectively, during any instant in which there is a transition on δ [7]. The configuration definition can be extended to multiple tapes using the corresponding cursors.

A deterministic Turing Machine is a Turing Machine that has only one next action for each step defined in the transition function [6], [4]. However, a non-deterministic Turing Machine can contain more than one action defined for each step of the program, where this program was no longer a function but a relation [6], [4].

A complexity class is a set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [2]. There are four complexity classes that have a close relationship with the previous concepts and are represented as P, UP, EXP and NP. In computational complexity theory, the class P contains the languages that are decided by a deterministic Turing Machine in polynomial time [6]. The class UP has all the languages that are decided in polynomial time by a non-deterministic Turing machines with at most one accepting computation for each input [10]. The complexity class EXP is the set of all decision problems solvable by a deterministic Turing machine in $O(2^{p(n)})$ time, where $p(n)$ is a polynomial function of n. The class NP contains the languages that are decided by a non-deterministic Turing Machines in polynomial time [4]. Problems that are $EXP-complete$ might be thought of as the hardest problems in EXP. We do know that $EXP-complete$ problems are not in P: it has been proven that these problems cannot be solved in polynomial time, by the time hierarchy theorem [7].

On the other hand, a language $L \in NP$ if there is a polynomial-time decidable, polynomially balanced relation R_L such that for all strings x: there is a string y with $R_L(x, y)$ if and only if $x \in L$ [7]. The function problem associated with L is the following computational problem: given x, find a string y such that $R_L(x, y)$ if such a string exists; if no such string exists, return "*no*" [7]. The class of all function problems associated as above with languages in NP is called FNP [7]. The resulting class from FNP is the class FP which represents all function problems that can be solved in polynomial time [7]. We also could define $FEXP$ as the complexity class of function problems associated with languages in EXP.

The P versus NP problem is to know whether P is equal to NP or not. This would be equivalent to prove whether FP is equal to FNP or not. A one-way function f is a function from strings to strings, one-to-one, for all input x we have $\mid x \mid^{\frac{1}{k}} \leq \mid f(x) \mid \leq \mid x \mid^{k}$ for some $k > 0$ and f is in FP but f^{-1} is not [7]. It holds the following statement: $P = UP$ if and only if there are no one-way functions [7]. If one-way functions exist, then $P \neq NP$ [5]. The existence of one-way functions is still an open conjecture.

3. Results

Lemma 3.1. *Every language $L_{exp} \in EXP$ is decided by a deterministic Turing Machine M_{exp} that has only one tape and always accepts in the configuration $(\text{“}yes\text{”}, \rhd, x\$z)$ when $x \in L_{exp}$ and z is another remaining string of the accepting computation that will be at most polynomially longer or shorter than the string x.*

Every Turing Machine could be transformed into another Turing Machine of one tape which has a polynomial time in relation with the running time of the original [7]. Therefore, the deterministic Turing Machine that decides L_{exp} could be of one tape. This one-tape deterministic Turing Machine can be transformed into two-tapes deterministic Turing Machine that receives the input in the first tape. This new Turing Machine will copy the input in the second tape and there, it will simulate the original Turing Machine of one tape. When the simulation of the original Turing Machine accepts, it will copy, concatenated with a blank symbol at the end of the first tape, the content of the second tape until a length which is at most polynomially longer or shorter with the size of x in a constant exponent and if the content is very short or empty, then we fill it with 1 symbols. Next, we remove the content in the second tape that was not copied on the first tape.

Finally, it will set the cursors in the start symbols of each tape and halt in the state of acceptance. In case of rejection, the two-tapes deterministic Turing Machine will reject too. This new Turing Machine can be transformed into a one-tape Turing Machine M_{exp} complying with the Lemma 3.1.

Definition 3.2. *For every language $L_{exp} \in EXP$, we could invert the deterministic Turing Machine M_{exp} of Lemma 3.1 changing the state of acceptance with the initial state and reversing the transition function of M_{exp}. In this way, we would create a new non-deterministic Turing Machine N_{exp}. We are going to define the rejection state in N_{exp} in the following way: for every q state in the set of states of N_{exp} and every σ symbol of its alphabet, then $\delta(q, \sigma) = (\text{“}no\text{”}, \sigma, -)$, where δ will be the program of N_{exp}. The non-deterministic Turing Machine N_{exp} will simulate the behavior of M_{exp} moving backwards.*

We start executing $N_{exp}(x\$z)$ from the initial configuration $(s, \triangleright, x\$z)$ which corresponds to the configuration of acceptance $(\text{“}yes\text{”}, \triangleright, x\$z)$ on M_{exp}.

Definition 3.3. *Let $config(x)$ be a configuration that belongs to the accepting computation of $x \in L_{exp}$ for some language $L_{exp} \in EXP$ on the deterministic Turing Machine M_{exp} of Lemma 3.1 and is at most polynomially longer or shorter than the string x.*

We could obtain the configuration by the deterministic Turing Machine M_{exp} of Lemma 3.1 in the execution of $M_{exp}(x)$ or by the non-deterministic Turing Machine N_{exp} of the Definition 3.2 in the execution of $N_{exp}(x\$z)$.

Theorem 3.4. *For every language $L_{exp} \in EXP$ and is not in P, we obtain the configuration $config(x)$ by an algorithm that runs forward from the initial configuration in a polynomial time if and only if we cannot obtain the configuration $config(x)$ by an algorithm that runs backward from the acceptance configuration in polynomial time. The initial and acceptance configurations belong to the deterministic Turing Machine M_{exp} of Lemma 3.1 with an input $x \in L_{exp}$.*

If we obtain the configuration $config(x)$ by the execution of $M_{exp}(x)$ in a polynomial time, then that means from the configuration $config(x)$ until the state of acceptance there is an exponential amount of steps, because $L_{exp} \in EXP$ and is not in P. Therefore, any algorithm that runs backward from the acceptance configuration with the input $x\$z$ until the configuration $config(x)$ will be exponential due to z is at most polynomially longer or shorter than the strings x. If we obtain the configuration $config(x)$ by the execution of $N_{exp}(x\$z)$ in a polynomial time, then that means from the configuration $config(x)$ until the state of acceptance on M_{exp} with x there will be a polynomial amount of steps, because $config(x)$ is at most polynomially longer or shorter than the strings x and z. Therefore, there will not be any polynomial time algorithm that runs forward from the initial configuration with x until $config(x)$ due to $L_{exp} \in EXP$ and is not in P.

Hypothesis 3.5. *The one-way functions exist.*

We are going to assume the Hypothesis 3.5 is true until the end of this work where we prove that is false.

Lemma 3.6. *There are at least two one-way functions.*

If we have at least one function f_* that is one-way, then we could build for every function f, that is from strings to strings, one-to-one, for all input x we have $\mid x \mid^{\frac{1}{k}} \leq \mid f(x) \mid \leq \mid x \mid^k$ for some $k > 0$ and f and f^{-1} are in FP, another function f_{**} that is one-way too. There are many functions such as the identity function which comply with the characteristics of f. The one-way function f_{**} will be defined by the composition of functions $f_*(f(x))$ for any input x. We already know that $f_*(f(x))$ is in FP, but to compute f_{**}^{-1} is necessary to execute f_*^{-1}, because $x = f^{-1}(f_*^{-1}(f_*(f(x))))$. Therefore, f_{**}^{-1} is not in FP, and then, f_{**} complies with all the characteristics in the definition of a one-way function.

Definition 3.7. *For each two one-way functions f_* and f_{**}, we could define a new function problem F_{***}, such that for every input x the function problem will return the result of $f_{**}^{-1}(f_*(x))$. If x and $f_*(x)$ are not defined in the domains of f_* and f_{**}^{-1} respectively, then F_{***} returns "no".*

Lemma 3.8. *$F_{***} \in FEXP$ and is not in FP.*

This is possible due to f_{**}^{-1} is not in FP.

Lemma 3.9. *The function problem F_{***} has an associated language L_{***} such that $L_{***} \in EXP$ and is not in P.*

This result is a consequence of the definition of function problem that we explain in the Theory section.

Important 3.10. *With the language L_{***}, we will put the remaining string z of the accepting computation in the configuration ("yes", $\triangleright, x\$z$) when $x \in L_{***}$ on the deterministic Turing Machine M_{exp} of Lemma 3.1 as the output of F_{***} for x.*

This will be the key in our proof.

Lemma 3.11. *Each one-way function f_* and f_{**} has a different deterministic Turing Machine M_* and M_{**} that has only one-tape and halts with polynomial time in the state of halting for the input x with the configuration $(h, \triangleright, y)$ when $f_*(x) = y$ and $f_{**}(x) = y$ respectively.*

The functions f_* and f_{**} are in FP, and thus, following the arguments of Lemma 3.1, we could build any of these deterministic Turing Machines M_* or M_{**} for each function in a similar way, that is creating a new input tape, removing the content of the other tapes at the end and changing the new Turing Machine into a one-tape deterministic Turing Machine.

Lemma 3.12. *In the language L_{***}, we could have a configuration which belongs to the accepting computation of $x \in L_{***}$ on the deterministic Turing Machine M_{exp} of Lemma 3.1 that is equal to the configuration in the halting state on M_* in the execution of $f_*(x)$, but with a different state. We will denoted this configuration as $config_*(x)$.*

This is an implication of the function problem F_{***} associated with L_{***}, because F_{***} executes $f_*(x) = y$ and then $f_{**}^{-1}(y)$. Hence, we could first simulate the execution of the one-tape deterministic Turing Machine M_* with x for f_* on M_{exp} and later $f_{**}^{-1}(y)$.

Theorem 3.13. *The Hypothesis 3.5 is false, and therefore, there are no one-way functions.*

We show we could obtain $config_*(x)$ on the deterministic Turing Machine M_* with x for f_*, but with a different state. Besides, we could obtain $config_*(x)$ on the deterministic Turing Machine M_{**} with $f_{**}^{-1}(f_*(x))$ for f_{**}, but with a different state. Indeed, the string $f_{**}^{-1}(f_*(x))$ will be the output of F_{***} for the input x that will be equal to the string z on the configuration of acceptance on M_{exp}. However, the state of the configuration $config_*(x)$ could be always the same, because we could delimit on M_{exp} when we finish through $f_*(x) = y$ and start with $f_{**}^{-1}(y)$ and then we could assign a single state for this step. For that reason, we could affirm that we can always obtain the configuration $config_*(x)$ by a polynomial time algorithm in a forward and backward running way from the initial and acceptance configurations which belong to the deterministic Turing Machine M_{exp} of Lemma 3.1 with an input $x \in L_{***}$. Nevertheless, this is a contradiction with Theorem 3.4. In conclusion, if we apply the reductio ad absurdum method, we have the Hypothesis 3.5 is false, and therefore, there are no one-way functions.

Lemma 3.14. *$P = UP$.*

This is a direct consequence of Theorem above.

4. Conclusions

This result shows in a formal way that many currently mathematically problems can be solved efficiently such as the integer factorization of two large primes. In this way, it proves that is not safe many of the encryption and authentication methods such as the public-key cryptography. It could be the case of $P = NP$ or $P \neq NP$, even though there are no one-way functions. However, we prove that $P = UP$.

Acknowledgement

I thank my mother Iris Delgado for her support and confidence.

References

1. Stephen A. Cook, *The complexity of theorem proving procedures*, Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing (STOC'71), ACM Press, 1971, pp. 151–158.
2. Thomas H. Cormen, Charles Eric Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms, second edition*, MIT Press, 2001.
3. Lance Fortnow, *The status of the P versus NP problem*, Communications of the ACM **52** (2009), no. 9, 78–86.
4. M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of np-completeness (series of books in the mathematical sciences)*, first edition ed., W. H. Freeman, 1979.
5. O. Goldreich, *The foundations of cryptography - volume 1, basic techniques*, Cambridge University Press, 2001.
6. Harry R. Lewis and Christos H. Papadimitriou, *Elements of the theory of computation (2. ed.)*, Prentice Hall, 1998.
7. Christos H. Papadimitriou, *Computational complexity*, Addison-Wesley, 1994.
8. M. Sipser, *Introduction to the theory of computation*, International Thomson Publishing, 1996.
9. Alan M. Turing, *On computable numbers, with an application to the entscheidungsproblem*, Proceedings of the London Mathematical Society **42** (1936), 230–265.
10. Leslie G. Valiant, *Relative complexity of checking and evaluating.*, Inf. Process. Lett. **5** (1976), no. 1, 20–23.

Datys, Playa, Havana, Cuba
E-mail address: vega.frank@gmail.com