EasyChair Preprint

№ 7282

The Existence of One-Way Functions

Frank Vega

January 5, 2022

The Existence of One-Way Functions

Frank Vega[1][0000−0001−8210−4126]

CopSonic, 1471 Route de Saint-Nauphary 82000 Montauban, France
vega.frank@gmail.com
https://uh-cu.academia.edu/FrankVega

Abstract. Under the assumption that there exist one-way functions, then we obtain a contradiction following a solid argumentation and therefore, one-way functions do not exist by contraposition. Hence, function problems such as the integer factorization of two large primes can be solved efficiently. In this way, we prove that is not safe many of the encryption and authentication methods such as the public-key cryptography. It could be the case that $P = NP$ or $P \neq NP$, even though there are no one-way functions. However, this result proves that $P = UP$.

Keywords: complexity classes · one-way function · polynomial time · exponential time.

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 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. The existence of one-way functions is still an open conjecture.

2 Ancillary 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 nondeterministic Turing Machine can contain more than one action defined for each step of the program, where this program is 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].

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].

The P versus NP problem asks 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].

3 Results

Definition 1. *We denote every language in EXP that is not in P as L_{exp}. Moreover, we denote M_{exp} as the one-tape deterministic Turing Machine which decides L_{exp}.*

Lemma 1. *Every language L_{exp} can be actually decided by some one-tape deterministic Turing Machine M_{exp}, such that for every element $x \in L_{exp}$ the Turing Machine M_{exp} will accept in the configuration ("yes", $\triangleright$, x).*

Proof. Every Turing Machine of multiple tapes could be transformed into another Turing Machine of one tape which has a polynomial time difference in relation with the running time of the original one [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 delete the content in the second 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 again into the one-tape Turing Machine M_{exp} according to the Lemma 1. □

Definition 2. *We call $config(x)$ as any configuration which belongs to the accepting computation of some input $x \in L_{exp}$ on the deterministic Turing Machine M_{exp} of Lemma 1 and $config(x)$ complies with the following conditions:*

1. *The configuration $config(x)$ is at most polynomially longer than the corresponding input $x \in L_{exp}$.*
2. *We can compute the execution of $M_{exp}(x)$ from the configuration $config(x)$ until the state of acceptance with the string $x \in L_{exp}$ using only a polynomial amount of steps in relation with the size of x.*

Definition 3. *We denote $F_{L_{exp}}$ as the function problem of finding the configuration $config(x)$ of the Definition 2 for some input $x \in L_{exp}$.*

This definition will help us to state the following theorem.

Theorem 1. *For every language L_{exp}, the function problem $F_{L_{exp}}$ is not in FP.*

Proof. $F_{L_{exp}}$ is not in FP, because if we could find the configuration $config(x)$ which belongs to the accepting computation of some input $x \in L_{exp}$ in polynomial time, then we could simulate $M_{exp}(x)$ in polynomial time by reaching $config(x)$ in polynomial time with x, accepting in the following polynomial steps in relation with the size of x and checking if the final configuration is ("yes", $\triangleright$, x). However, this is not possible, because $L_{exp} \in EXP$ is not in P. □

Definition 4. *We call $fake(x)$ as any configuration for the input x on the deterministic Turing Machine M_{exp} of Lemma 1 and $fake(x)$ complies with the following conditions:*

1. *The configuration $fake(x)$ is at most polynomially longer than the corresponding input x.*
2. *We can compute the execution of $M_{exp}(x)$ from the configuration $fake(x)$ until the state of acceptance with the string x using only a polynomial amount of steps in relation with the size of x.*
3. *x is not in L_{exp}*

Theorem 2. *For every language L_{exp}, there could be many configurations $fake(x)$ for some inputs x when x does not belong to L_{exp}.*

Proof. We could invert the deterministic Turing Machine M_{exp} 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.

In this simulation, we are going to interrupt the usual exponential execution of $N_{exp}(x)$ just in the first $\mid x \mid^2$ steps for example where $\mid x \mid$ is the size of some input x when x does not belong to L_{exp}, and thus, we start executing N_{exp} from the initial configuration $(s, \triangleright, x)$ until some candidate configuration $fake(x)$. The configuration $fake(x)$ is at most polynomially longer than the corresponding string x, because from the initial configuration we cannot add more than $\mid x \mid^2$ symbols until the candidate configuration. In this way, we can compute the execution of $M_{exp}(x)$ from the configuration $fake(x)$ until the state of acceptance with the string x using only a polynomial amount of steps. □

Theorem 3. *For every language L_{exp}, the function problem of $F^{-1}_{L_{exp}}$ is not in FP too.*

Proof. We could find $x \in L_{exp}$ in polynomial time if we have the configuration $config(x)$ as input. However, the function problem $F^{-1}_{L_{exp}}$ should return “*no*” for the configurations $fake(y)$. However, the principal difference between $config(x)$ and $fake(y)$ is that x is in L_{exp} and y is not. Therefore, whether $F^{-1}_{L_{exp}}$ return “*no*” or not, it would be equivalent to decide the elements of the language $L_{exp} \in EXP$, but L_{exp} is not in P. Then, the Theorem is true. □

Theorem 4. *There are no one-way functions.*

Proof. We are going to assume that exists a function f that is one-way. We could define a new function problem as the composition of functions $f(f^{-1}(x))$ for the strings x in the domain of f^{-1} and in case of x is not in that domain, then the function problem return “*no*”. That function problem is associated with a language $L_{exp} \in EXP$ which is not in P, because f^{-1} is not in FP. Indeed, $x \in L_{exp}$ if and only if $f(f^{-1}(x)) = x$.

On the other hand, for the language $L_{exp} \in EXP$, the deterministic Turing Machine M_{exp} of Lemma 1 could be simulated by two deterministic Turing Machines $M_{f^{-1}}$ and M_f of one-tape which simulate the functions f^{-1} and f respectively. The running of M_{exp} with $x \in L_{exp}$ consist of: First, it will execute $M_{f^{-1}}(x)$ and next, if there is no rejection, it will continue the execution of M_f from the halting configuration of $M_{f^{-1}}(x)$. The final configuration for every element $x \in L_{exp}$ on the deterministic Turing Machine M_{exp} will be $(\text{“}yes\text{”}, \triangleright, x)$, because the halting configuration in M_f could be $(h, \triangleright, x)$ and we could replace the state of halting in M_f by the state of acceptance in M_{exp}.

In this way, the halting configuration of $M_{f^{-1}}(x)$ for some input $x \in L_{exp}$ has all the properties of a $config(x)$ in the Definition 2, because the function f is one-way. Hence, we could define the function problem $F_{L_{exp}}$ of finding the halting configuration of $M_{f^{-1}}(x)$ for some input $x \in L_{exp}$. However, $F^{-1}_{L_{exp}}$ would be in FP, because f is in FP, but this is a contradiction with the Theorem 3. Therefore, there are no one-way functions by contraposition. □

Lemma 2. $P = UP$.

Proof. This is a direct consequence of Theorem 4. □

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, we prove 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 $P = UP$.

Acknowledgments

The author would like to thank Chenggang Lu for remind him this old approach and his mother, maternal brother and his friend Sonia for their support.

References

1. Cook, S.A.: The complexity of theorem proving procedures. In: Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing (STOC'71). pp. 151–158. ACM Press (1971)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Second Edition. MIT Press (2001)
3. Fortnow, L.: The status of the P versus NP problem. Communications of the ACM **52**(9), 78–86 (Sep 2009)
4. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences). W. H. Freeman, first edition edn. (1979)
5. Goldreich, O.: The Foundations of Cryptography - Volume 1, Basic Techniques. Cambridge University Press (2001)
6. Lewis, H.R., Papadimitriou, C.H.: Elements of the theory of computation (2. ed.). Prentice Hall (1998)
7. Papadimitriou, C.H.: Computational complexity. Addison-Wesley (1994)
8. Sipser, M.: Introduction to the Theory of Computation. International Thomson Publishing (1996)
9. Turing, A.M.: On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London Mathematical Society **42**, 230–265 (1936)
10. Valiant, L.G.: Relative complexity of checking and evaluating. Inf. Process. Lett. **5**(1), 20–23 (1976)