Effective enumeration of infinitely many programs that evade Formal Malware Analysis*

V. Liagkou[1,4], P.E. Nastou[5], P. Spirakis[2], and Y.C. Stamatiou[1,3]

[1] Computer Technology Institute and Press - "Diophantus", University of Patras Campus, 26504, Greece
[2] Department of Computer Science, University of Liverpool, UK and Computer Engineering and Informatics Department, University of Patras, 26504, Greece
[3] Department of Business Administration, University of Patras, 26504, Greece
[4] University of Ioannina, Department of Informatics and Telecommunications, 47100 Koatakioi Arta, Greece
[5] Department of Mathematics, University of the Aegean, Applied Mathematics and Mathematical Modeling Laboratory, Samos, Greece
e-mails: liagkou@cti.gr, pnastou@aegean.gr, P.Spirakis@liverpool.ac.uk, stamatiu@ceid.upatras.gr

Abstract. The formal study of computer malware was initiated in the seminal work of Fred Cohen in the mid 80s who applied elements of the Theory of Computation in the investigation of the theoretical limits of using the Turing Machine formal model of computation in detecting viruses. Cohen gave a simple but realistic, *formal*, definition of the characteristic actions of a computer virus as a Turing Machine that replicates itself and then proved that constructing a Turing Machine that recognizes viruses (i.e. Turing Machines that act like viruses) is impossible, by reducing the Halting Problem, which is undecidable, to the problem of recognizing a computer virus. In this paper we complement Cohen's approach along similar lines, based on Recursion Function Theory and the Theory of Computation. More specifically, after providing a simple generalization of Cohen's definition of a computer virus, we show that the malware/non-malware classification problem is undecidable under this new definition. Moreover, we show that to any formal system, there correspond *infinitely many*, *effectively constructible*, programs for which no proof can be produced by the formal system that they are either *malware* or *non-malware* programs. In other words, given any formal system, one can provide a procedure that generates, systematically, an *infinite number* of impossible to classify, within the formal system, programs.

1 Introduction

In this paper we investigate the problem of *classifying programs* either as *malware* or *non-malware* based on formal proof systems and their de-

* The work of the first, third and fourth coauthors was partially supported by the CyberSec4Europe project, funded by the European Union under the H2020 Programme Grant Agreement No. 830929

ductive procedures. Our goal is to study the deductive power of formal systems with respect to the problem of producing proofs that can characterize *all* computer programs either as *malware* or *non-malware*. Our work is based on the foundations of computability and recursive function theory which, essentially, study problems with respect to their theoretical solvability based on the universal Turing Machine model of a mechanical and effective computation.

Formal proofs about the impossibility of detecting, in a *systematic* (i.e. algorithmic) and *general* way, malicious entities, such as Malware in our case, already exist for a long time for a very important category of such entities, the *computer viruses* or *malware* in general. A virus is a malicious program, a Turing Machine formally, that operates with an aim to *replicate in* other programs (Turing Machines), thus spreading the infection. The formal study of computer programs which act as viruses and their algorithmic detection was initiated in the seminal work of Fred Cohen in the mid 80s (see [1, 2]). Cohen starts with a simple, *formal*, definition of the characteristic actions of a virus. Then, he proceeds to prove that constructing a Turing Machine that *recognizes* viruses (technically, other Turing Machines that act like viruses) is impossible.

More specifically, Cohen defined a virus to be a program, or Turing Machine, that simply copies itself to other programs, or more formally, injects its transition function into other Turing Machines' transition functions (see Definition 1 in Section 2) replicating, thus, itself indefinitely. Then, he proves that the problem of deciding whether a given Turing Machine halts on a given input, i.e. deciding the language L_u, can be reduced to the problem of deciding whether a given Turing Machine is a virus, i.e. $L_u = \{< M, w > | w \in L(M)\}$ is *reduced* to $L_v = \{< M > | M \text{ is a virus } \}$. Since L_u is undecidable, so must be L_v and, thus, it is in principle impossible to detect a virus or else we could decide L_u which is, provably, undecidable.

Following Cohen's paradigm, we will propose a rather restricted (so as to be amenable to a theoretical analysis) but reasonable and precise definition of malware. To this end, we also deploy the *Turing Machine* theoretical model of an effective computational procedure to model malware programs. Thus, a malware is a program, i.e. a Turing Machine, that executes, at some point of its operation, *at least one* action from a specific set of actions that characterize malware behaviour (these actions are called *states* in the Turing Machine definition). We remark that simply *locating* the actions in a program through, e.g. syntactic analysis, is not considered to manifest malware behaviour, in the proposed model, only their

actual execution is considered to manifest such a behaviour. Admittedly this is a, rather, restricted malware model since malicious behaviour can be complex, e.g. in DoS (Denial of Service) attacks or may include several, combined, steps or Turing Machine states (e.g. Ransomware, which executes sequences of malicious actions to encrypt files on a victim computer). However, we chose this simpler model in order to be able to benefit from the rich and deep results of the Theory of Computation and, thus, provide some first results based on an established and mature computational model and scientific discipline. Consequently, our goal is to use this, rather restricted but theoretically manageable and plausible, malware model in order to obtain a malware undecidability result similar to Cohen's, i.e. there is no algorithm (Turing Machine) that can detect, systematically, all malware programs that fall under this definition.

The theoretical undecidability of the malware/non-malware classification problem means that, in general, no algorithm exists that can take as input a program and decide whether it is a malware or not. Our next step, in this paper, is to investigate whether it is possible to *effectively* demonstrate one of the programs which are not amenable to classification. In other word, our goal is to see whether it is possible to *construct* a specific Turing Machine whose classification status as either malware or non-malware cannot be decided. In the context of malware, such a Turing Machine would provide evidence of, potentially, hard or impossible to detect malware. To investigate this possibility, we turn to formal systems and, in particular, formal systems powerful enough to enable statements about Turing Machines, such as statements that state whether they halt, given a particular input, or not. We show that to each *consistent* formal system, there corresponds an *infinite*, *recursively enumerable* set of Turing Machines for which there exists no proof, in the formal system, that they are malware *and* there exists no proof, in the same formal system, that they are not malware. This is a humble example of a nature similar to Gödel's famous, groundstaking, result about the *incompleteness of consistent formal systems* in the sense that for each such system, there exist statements, of a self-referential nature, that neither themselves nor their negations can be proved within the formal system. In addition, these *self-referential* statements are, indeed, *true* (but not provable), *if and only if* the formal system is consistent (see, e.g., [9] for an accessible coverage of Gödel's Incompleteness Theorems and their consequences for formal systems). Inspired by this important consequence of the incompleteness result, we show that the infinitely many Turing Machines whose malware/non-malware status is impossible to prove within a consistent

formal system, are actually *non-malware* but it is *impossible* to prove it within the formal system by the, purely. formal procedures allowed within the formal system. In other words, we show that these Turing Machines are *non-malware* if and only if the formal system used to classify them as malware or non-malware *is* consistent.

Before we proceed, we should remark that theoretical impossibility *does not* imply impossibility in practice since Turing Machines are idealistic models of computers with unlimited computational resources. However, the *finite* nature of real computers and programs renders all undecidable problems decidable by simple (but highly inefficient in practice) brute force approaches. Thus, theoretical impossibility results may not translate, readily, into impossibility results in practice.

2 Foundations of Computation Theory

We will assume a basic level of familiarity with the fundamental concepts and results of *Recursive Function Theory*. However, for completeness, we will briefly review Turing Machines and the basic results of the Theory of Computation and Recursive Function theory. In doing so, we extend in a straightforward way the standard Turing Machine model in order to model malware activity. With respect to the presentation, we follow the exposition in the excellent, classic, book on the subject by Hoprcoft and Ullmann [5]. Before providing the details, we note that there are other, more practical, computational models that could be employed but since the theory we deploy concerns the classical Turing Machine model, we decided to base our results on this model for simplicity. We feel, however, that the results can be extended, with some effort, to other computational models which are more realistic.

Definition 1 (Turing Machine). *A Turing Machine is an octuple, defined as* $M{=}(Q, Q_{Mal}, \Sigma, \Gamma, \delta, q_0, B, F)$*, where* Q *is a finite set of states,* Γ *is a finite set called the tape alphabet, where* Γ *contains a special symbol* B *that represents a blank,* Σ *is a subset of* $\Gamma - \{B\}$ *called the input alphabet,* δ *is a partial function from* $Q{\times}\Gamma$ *to* $Q{\times}\Gamma{\times}\{L, R\}$ *called the transition function,* $q_0{\in}Q$ *is a distinguished state called the start state,* $F \subset Q$*,* $q_0 \notin F$*, is a set of final states, and* $Q_{Mal} \subset Q$*,* $Q_{Mal} \cap F = \emptyset$*, is a distinguished set of states linked to Malware behaviour. We assume that transitions from states in* Q_{Mal} *do not change the Turing Machine's tape contents, i.e. they are purely interactions with the external environment of the Turing Machine and can affect only the environment.*

We should remark that there are three basic assumptions in the definition of a Turing Machine:

1. The tape can be extended on a need basis, i.e. each time the machine needs more cells (memory) on the tape, which is assumed to be automatically extended. In this respect, the memory of a Turing Machine is, virtually, unlimited.
2. Computational steps, i.e. reading of a tape cell, change of state and advance of the tape head, are executed instantaneously.
3. There are no operation errors during the operation of the machine.

Thus, the Turing Machine is an idealized formal model of a real computer that abstracts away from construction, technology, and speed details not relevant to the *fundamentals* of mechanical or algorithmic computation in order to allow us to explore the *theoretical* limits of machines in solving problems.

A Turing Machine can be viewed either as a *language* acceptor or a *function* calculator. The language *accepted* by a Turing Machine M, denoted by $L(M)$, is the set of strings in Σ^* that when given as input to M, lead it to an acceptance state, i.e. a state in set F. In recursive function theory, these languages are also called *recursively enumerable*. In general, without loss of generality, we can assume that the Turing Machine, when it accepts an input string, it also halts i.e. there are no next steps after an accepting state. In this paper, we will not use Turing Machines that compute functions. However, we should remark, that there is no loss of generality by confining ourselves to Turing Machines as language acceptors (see, for instance, the discussion in [5]). Thus, for our purposes, Turing Machines operate as language acceptors (see below).

In Figure 1 we see the structure of a Turing Machine, according to Definition 1, and in Table 1 we see a sample Turing Machine computation.

	q_0	q_1	q_2	q_3	q_4	q_5	q_6
0	(q_1,#,Δ)	(q_1,0,Δ)	(q_3,1,A)	(q_3,0,A)	(q_4,0,A)	(q_5,#,Δ)	- (stops)
1	(q_5,#,Δ)	(q_2,1,Δ)	(q_2,1,Δ)	(q_3,1,A)	(q_4,#,A)	(q_5,#,Δ)	- (stops)
#	- (hangs)		(q_4,#,A)	(q_0,#,Δ)	(q_6,0,Δ)	(q_6,#,Δ)	- (stops)

Table 1. Operation of a TM

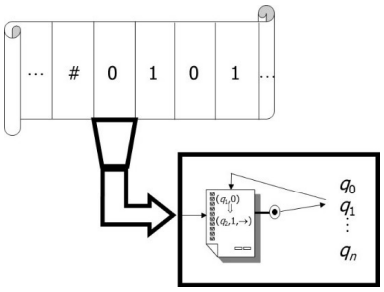

Fig. 1. The Turing Machine computation model

Thus, a Turing Machine is, essentially, a theoretical model of a real computer that allows us to study the *power* of mechanical, or *algorithmic*, computation and its *limits*. As a consequence, we can also deepen our understanding about which problems can be solved in principle by mechanical computations.

With respect to the evolution and termination of the computation of a Turing Machine, we have the following three possibilities:

1. The machine halts in a *final* state. Then the input string is accepted (i.e. it belongs to the language accepted by the Turing Machine). There is no loss of generality in assuming that the Turing Machine halts (i.e. it has no next move) whenever the input is accepted.
2. The machine halts in a *non-final* state. Then the input is rejected (i.e. it does not belong to the language accepted by the Turing Machine).
3. The machine does not halt at all, i.e. it runs for ever (it has entered an "infinite loop", as computer engineers say). Then certainly the input string is never accepted, i.e. it does not belong to the language accepted by the Turing Machine, but there is no way (as we will see below) to decide that the computation will not ever terminate (it is the famous, undecidable, *Halting problem*).

With respect to notation, for a given Turing Machine M we denote by $< M >$ its *code*, i.e. an encoding of its description elements as stated in Definition 1 using any fixed alphabet, usually the alphabet $\{0, 1\}$ (binary system) which since the inception of computing machines was the alphabet of choice due to its simplicity and efficiency in representing it with electronic states (two such states suffice). The details of such an encoding can be found in, e.g., [5] but it is really much like the representation of a program in machine code or assembly, which are the native programming languages executed by processing units in modern computers.

One of the major results of Turing's seminal work on computability was the existence of a *universal Turing Machine*, that is a Turing Machine that takes as input strings which represent other Turing Machines and their inputs and simulates (executes) them producing their results on their behalf. This is actually what a modern computer does, taking as an input program descriptions along with their inputs and producing their outputs by executing them (see, e.g., [5] for the encoding details).

The main outcome of Turing's pioneering work was, the formalization of what a mechanical procedure is using the Turing Machine model. This work gave rise to the computability theory, that classifies problems according to whether they can be solved, *in principle*, by mechanical procedures or Turing Machines. One of the major results of Alan Turing was the proof that there exist problems that Turing Machines cannot solve. For instance, the first problem (theoretically, there are infinitely many such problems) what was shown to be unsolvable by Turing Machines (i.e. algorithms) is the, so called, *Halting problem*:

The Halting Problem
Input: A string $x =< M, w >$ which is actually the encoding (description) of a Turing Machine <M> and its input w.
Output: If the input Turing Machine M halts on w, the output is True. Otherwise, the output is False.

The language corresponding to the Halting problem is $L_u = \{< M, w > | w \in L(M)\}$. In other words, the language L_u contains all possible *Turing Machine-input* pair encodings $< M, w >$ such that w is accepted by M. This is why L_u is also called *universal language* since the problem of deciding whether a given Turing Machine M accepts a given input w is equivalent to deciding whether $< M, w > \in L_u$. Note that L_u is accepted by a Turing Machine, the universal one denoted by M_u, that simulates M on w. If $w \in L(M)$, then the universal Turing Machine will accept the pair $< M, w >$. However, if the opposite holds true, then we only know that M_u will not terminate in a final state. What we do not know, however, is whether M_u will ever stop (in a non-final state of course). Languages for which Turing Machines exist that accept them and always halt are called *recursive* or decidable.

The language L_u was the first language proved to be non-recursive or undecidable by Turing, meaning that it is impossible to decide algorithmically whether a given program halts or not on a given input. The proof relies on a clever Cantor diagonalisation, self-reference based, argument over all possible Turing Machines. Such arguments lie in the heart

of mathematical logic as well as theoretical computer science for proving *impossibility* or *non-existence* arguments.

3 Recursive Function Theory

We should stress the fact that Computation and Recursive Function Theory are *computation model independent*. The model of computation can be any reasonable model such as the Turing Machine (see Section 2), the λ-terms (in the λ-Calculus), μ-recursive functions as well as, in a more practical perspective, any real computer programming language. It is not hard to prove that the theoretical computational power of all these computation models is the same, if we disregard efficiency issues.

In addition, the computational procedures or *programs* that can be written in any computation formalism or real programming language can be enumerated *effectively* (see, e.g., [5]). This means that there exists an algorithm which can list all the programs in a sequence, so that *all* programs appear at some point of the enumeration procedure.

A well known result from computability theory states that the number of arguments in a function (program) is not important, for computability theory, since arguments can, always, be embedded, in an easy way, in the program itself, reducing, in this way, the number of arguments. This is formalized in the following result, which is a simplified form of *Kleene's S_{mn}-theorem* (see [6]), for functions of two arguments.

Theorem 1 (Kleene's S_{mn}-theorem - simplified form, for two-input functions). *Let $g(x, y)$ be a partial recursive function. Then, there is a total recursive function σ of one variable, such that $f_{\sigma(x)}(y) = g(x, y)$ for all x and y. That is, if $\sigma(x)$ is considered as the integer (code) representing some TM M_x then $f^{(1)}_{M_x} = g(x, y)$.*

In addition, we will need another result from recursive function theory, namely the *Recursion Theorem* which states that every total recursive function that maps Turing Machine indices on Turing Machine indices has a fixed point. We state, formally, this theorem below as Theorem 2.

In what follows, we fix a formal system $\mathcal{F}$, e.g. such as *Peano's Arithmetic*, which can, directly, express statements about natural numbers. Our departure point is the, already, known fact that, given such a formal system, which we will denote by $\mathcal{F}$, we can, effectively, construct a Turing Machine M with the following property: there is no proof in $\mathcal{F}$ that M, when started on a specific input, halts and there is no proof that M, when started on a specific input, does not halt. The proof can be found in [5],

Chapter 8. We will denote by $M_{\mathcal{F}}$ such a Turing Machine (there may be several with this property) whose halting status in unprovable in $\mathcal{F}$.

Naturally, the details of $M_{\mathcal{F}}$ depend on $\mathcal{F}$ but the important issue, in our context, is that M_F can be constructed *effectively*, i.e. algorithmically, but, not necessarily, *efficiently* i.e. fast. In what follows we will present the proof of this fact since some of its elements are crucial in the presentation of our ideas. We, first, state the *Recursion Theorem* along with its proof (based on [5]) since it is crucial for our arguments that follow in Section 4.

Theorem 2. *For any total recursive function σ there exists an x_0 such that $f_{x_0}(x) = f_{\sigma(x_0)}(x)$, for all x.*

Proof. For each integer i, we construct a Turing Machine that when given input x it computes $f_i(i)$. Then, it simulates the $f_i(i)$th Turing Machine on x. Let $g(i)$ be the index of the constructed Turing Machine. By definition

$$f_{g(i)}(x) = f_{f_i(i)}(x) \tag{1}$$

for all x. Note that $g(i)$ is a total function, i.e. defined everywhere (and, thus, the TM that computes it always halts) even if $f_i(i)$ is not defined, i.e. f_i does not halt with input i. Let j be an index of the function σg, i.e. j is the index (encoding) of a TM that, when given i as input, computes $g(i)$ and then applies the function σ on $g(i)$. Thus, for $x_0 = g(j)$ the following is derived, after some manipulations:

$$f_{x_0}(x) = f_{\sigma(x_0)}(x) \tag{2}$$

for all x. Therefore, x_0 is a fixed point of the mapping σ, i.e the TMs x_0 and $\sigma(x_0)$ compute the same function. □

Based on the Recursion Theorem, the following, central to our approach in Section 4, is proved in [5]:

Theorem 3. *Given a formal system $\mathcal{F}$, we can construct a Turing Machine for which no proof exists in $\mathcal{F}$ that it either halts or does not halt.*

Proof. Given $\mathcal{F}$, we construct a Turing Machine M that computes a function, $g(i, j)$, of two inputs as follows:

$$g(i,j) = \begin{cases} 1, \text{ if there is a proof in F that} f_i(j) \text{ is not defined} \\ \quad \text{(i.e. does not halt) or, in other words if there is} \\ \quad \text{a proof that the } i\text{th Turing Machine does not} \\ \quad \text{halt, given input } j \\ \\ \text{undefined, otherwise} \end{cases} \tag{3}$$

The Turing Machine M works by enumerating proofs in $\mathcal{F}$. When a proof is found that states that the ith Turing Machine *does not* halt when given input j. Moreover, M can be designed so that if $g(i,j) = 1$ then it halts, otherwise it does not halt.

From the S_{mn}-theorem (see Theorem 1), there exists a total function σ on Turing Machine indices (i.e. codes) such that

$$f_{\sigma(i)}(j) = g(i,j). \tag{4}$$

From Recursion Theorem, we can construct an integer i_0 such that

$$f_{i_0}(j) = f_{\sigma(i_0)}(j) = g(i_0,j). \tag{5}$$

However, $g(i_0,j) = 1$ and it is, thus, defined if and only if there exists a proof in $\mathcal{F}$ that states that $f_{i_0}(j)$ is not defined. Therefore, if $\mathcal{F}$ is consistent, i.e. there can be no proofs of, both, a statement and its negation, then no proof can exist in $\mathcal{F}$ that the i_0-th Turing Machine either halts or does not halt when given a specific input j. □

The corollary that follows below is not stated in [5] but it is not hard to prove, as a consequence of Theorem 3.

Corollary 1. *For the i_0-th Turing maching, denoted by M_{i_0}, constructed in the proof of Theorem 3 and computing the function $f_{i_0}(j)$, it holds that it does not halt for every input j if and only if $\mathcal{F}$ is consistent.*

Proof. Let assume that $\mathcal{F}$ is consistent and that M_{i_0} halts for some input j_0. Then, since $\mathcal{F}$ is consistent, there can be no proof in $\mathcal{F}$ that M_{i_0} does not halt, for *any* input j. According, then, to the definition of $g(i,j)$ in Equation 3, Theorem 3, $g(i_0,j_0)$ is undefined. But since (see Theorem 3) $f_{i_0}(j_0) = g(i_0,j_0)$, we arrive at a contradiction since the left-hand side is defined and the right-hand side is undefined.

Let us assume, now, that M_{i_0} does not halt, for every input j. It follows, since $f_{i_0}(j) = g(i_0,j)$, that $g(i_0,j)$ is undefined for every j. Thus,

there is no proof in $\mathcal{F}$ that M_{i_0} does not halt on input j, for any j. Accordingly, $\mathcal{F}$ must be consistent since, otherwise, it can produce a proof, in $\mathcal{F}$, that M_{i_0} does not halt on input j, as everything follows from an inconsistent formal system, leading to a contradiction, since $g(i_0, j)$ is undefined for all j. □

4 Theoretical impossibility of a complete formal malware/non-malware program classification

In this section we give a simple formal definition of malware following and extending Cohen's ideas.

Definition 2. *(Formal Malware definition) A Malware is a Turing Machine that when executed will demonstrate a specific, recognizable, behavior particular to malware, as manifested by the* execution *(not simply the existence in the Turing Machine's description) of a specific sequence of actions, e.g. it will publish secret information about an entity, it will download information illegally etc., actions reflected by reaching, during its operation, states in the set Q_{Mal} (see Definition 1).*

This is similar to Cohen's definition of a virus since it characterizes Malware programs according to their *visible* or *manifested* behavior. We stress the word *"execution"* in order to preclude situations where a false alarm is raised for a "Malware" program which, merely, contains the states in Q_{Mal} without *ever* invoking them. Such programs, actually, operate normally without ever executing any actions characteristic to malware behavior.

The Malware Detection Problem
Input: A description of a Turing Machine (program).
Output: If the input Turing Machine behaves like Malware according to Definition 2 output True. Otherwise, output False.

More formally, if L_b denotes the language consisting of Turing Machine encodings $< M >$ which are Malware, according to Definition 2, then we want to decide L_b, i.e. to design a Turing Machine that, given $< M >$, decides whether $< M >$ belongs in L_b or not according to this definition.

Let Q_{Mal} be the set of actions which, when *executed*, manifest Malware behavior (see Definition 2). We will show that L_u is recursive in L_b. This implies that if we had a decision procedure for L_b then this procedure could also be used for deciding L_u which is undecidable. Thus, no decision procedure exists for L_b too.

In [7] the following was proved in a similar context:

Theorem 4 (Theoretical impossibility of detecting Malware). *The language L_b is undecidable.*

Proof. Our proof is similar to Cohen's proof about the impossibility of detecting viruses. Note that Rice's Theorem (see, e.g., [5]) is not applicable here since the Malware Detection Problem we consider does not involve properties of the *languages* accepted by Turing Machines but, rather, properties of their operation (i.e. reachability of a subset of their states, the *malware bahaviour* related states). In [7] we consider a detection problem for Turing Machines modeling Panopticons that involves properties of the *accepted languages* not their operation specifics.

Let $< M, w >$, with $M{=}(Q, Q_{\text{Mal}}, \Sigma, \Gamma, \delta, q_0, B, F)$ and $Q_{\text{Mal}} \subset Q$ the Malware states (see Definition 1), be an instance of the Halting problem. We will show how we can decide whether $< M, w >$ belongs in L_u or not using a hypothetical decision procedure for L_b, i.e. L_u is recursive in L_b.

Given $< M, w >$ we design a Turing Machine M^{u-b} that modifies the δ function of M so as when a final state is reached (i.e. a state in the set F of M) a transition takes place into a state in Q_{Mal} (any state suits our purpose). That is, M is a new Turing Machine M' containing the actions of M followed by actions (any of them) described by the states in Q_{Mal}. Now, M' is given as input the input w of M operating as described above.

Let us assume that there exists a Turing Machine M_b that decides L_b. Then we can give it M' as input. Suppose that M_b answers that $M' \in L_b$. Since Q_{Mal} was finally reached, this implies that M halted on w since M' initially simulated M on w. Then we are certain that M halts on w.

Assume, now, that M_b decides that M' is not Malware. Then a state in Q_{Mal} was never entered, which implies that no halting state is reached by M on w since Q_{Mal} in M' is reached only from halting states of M, which is simulated by M'. Thus, M does not halt on w.

It, thus, appears that M' is Malware if and only if M halts on w and, thus, we have shown that L_u is recursive in L_b. There is a catch, however, that invalidates this reasoning: if M *itself* can exhibit the Q_{Mal} linked Malware behavior in the first place. Then Malware behavior can be manifested, if states in Q_{Mal} are ever executed, without ever M reaching a final state that would trigger M' to enter a state in Q_{Mal}, by construction.

A solution to this issue is to *remove* Q_{Mal} from M, giving this new version to M^{u-b} to produce M'. Thus, we now have the equivalence M' is Malware if and only if M halts on w, completing the proof (see, also, [1]).

More formally, let $Q_{\text{Mal}} = \{q_{\text{Mal}_1}, q_{\text{Mal}_2}, \ldots, q_{\text{Mal}_l}\}$, $l = |Q_{\text{Mal}}|$ be the set of Malware states. We create a new set of "harmless" or "no operation" states $P' = \{q'_1, q'_2, \ldots, q'_l\}$ where q_{Mal_i} corresponds to q'_i and vice

versa. Then, we replace the states in Q_{Mal} by the corresponding states in P' everywhere in the definition elements of M and we also do the corresponding state changes in the δ function that defines the Turing Machine's state transitions. This transformation removes from a potential Malware the actions that *if* executed would manifest Malware behavior. We stress, again, the fact the mere *existence* of Malware actions in the definition of a Turing Machine is not considered Malware *action* if they are not *activated* at some point of its operation. With this last transformation, M' is a Malware if and only if M halts on w and, thus, L_u is recursive in L_b. □

We now turn to, actually, constructing a particular Turing Machine, which cannot be classified as malware or non-malware, by purely formal procedures, within any consistent formal system $\mathcal{F}$.

Theorem 5 (Malware/non-malware classification resistant programs). *Let $\mathcal{F}$ be a consistent formal system. Then we can construct a Turing Machine for which there is no proof in $\mathcal{F}$ that it behaves as malware and no proof that it does not behave as malware.*

Proof. Let $M_{\mathcal{F}}$ be a Turing Machine whose halting status on any given input j cannot be proven in $\mathcal{F}$ in either direction, i.e. "halts" or "does not halt". Such a Turing Machine exists by Theorem 3. This is a Turing Machine like M_{i_0} which was constructed in Theorem 3. A new Turing Machine, $M_{M_{\mathcal{F}}}=(Q, Q_{\text{Mal}}, \Sigma, \Gamma, \delta, q_0, B, F)$, of one input and $Q_{\text{Mal}} \subset Q$, the malware states, is constructed. It is composed of three parts. The first part is a *non-malware* Turing Machine, denoted by M_n the second part is $M_{\mathcal{F}}$, and the third part is a *malware* Turing Machine, denoted by M_w.

The construction details of these Turing Machines are not hard but they are tedious, thus we will provide a rather high level description. The construction of $M_{\mathcal{F}}$, given $\mathcal{F}$, has already been described in Section 2. With respect to M_n, it can be any Turing Machine that, simply, does not use any states in Q_{Mal}, e.g. a Turing Machine that computes a simple arithmetic function (see, for instance, Table 1 in Section 2). Finally, M_w executes, during its operation, at least one state in Q_{Mal}. It is not hard to construct such a Turing Machine, e.g. it can be a Turing Machine that simply, after leaving the start state, executes one more step involving a state in Q_{Mal} before halting (i.e. reaching a final state).

With respect to its operation, $M_{M_{\mathcal{F}}}$, first, activates the first part, i.e. M_n, which may ignore the input, say j, and operates with its non-Malware behavior, i.e. it *never* visits states in Q_{Mal} during its operation. Then $M_{\mathcal{F}}$ is activated with input j. By construction, $M_{\mathcal{F}}$ does not use states in

Q_{Mal}. Finally, the third part, i.e. M_w, starts operating, exhibiting Malware behaviour by visiting at least one state in Q_{Mal} during its operation.

Suppose, now, that a proof exists in $\mathcal{F}$ that $M_{M_\mathcal{F}}$ *is* a malware, i.e. it exhibits malware behaviour when activated by reaching states in Q_{Mal}. By the construction of $M_{M_\mathcal{F}}$, the only way to demonstrate malware behaviour is to activate its third part, i.e. M_w. This, in turn, can occur only if the second part, i.e. $M_\mathcal{F}$, halted on input j. Thus, the same proof that $M_{M_\mathcal{F}}$ *is* a malware, also, serves as a proof that $M_\mathcal{F}$ *halts* on input j.

Suppose, on the other hand, that a proof exists in $\mathcal{F}$ that $M_{M_\mathcal{F}}$ *is not* a malware, i.e. it *does not* exhibit malware behaviour when activated. By the construction of $M_{M_\mathcal{F}}$, this can happen only if M_w is never activated during the operation of $M_{M_\mathcal{F}}$. In turn, this can happen only if $M_\mathcal{F}$ *does not halt* on input j. Thus, again, the same proof that $M_{M_\mathcal{F}}$ *is not* a malware, also, serves as a proof that $M_\mathcal{F}$ *does not halt* on input j. □

From Theorem 5 we have the following corollary:

Corollary 2. *To any formal system $\mathcal{F}$, there correspond infinitely many, effectively constructible, Turing Machines for which there is no proof in $\mathcal{F}$ that they behave as malware and no proof that they do not behave so.*

Proof. Observe that in the *effective*, i.e. algorithmic, construction process described in Theorem 5, M_n can be *any of countably many infinite* Turing Machines that simply avoid the states in Q_{Mal} and M_w can be *any of countably many infinite Turing Machines* that do not visit states in Q_{Mal} during their operation. $M_\mathcal{F}$ stays fixed (it depends only on $\mathcal{F}$). □

Finally, in the same spirit with Corollary 1 for the Turing Machine M_{i_0} or $M_\mathcal{F}$ in the notation of Theorem 5, we prove the following about $M_{M_\mathcal{F}}$ which, actually, shows that $M_{M_\mathcal{F}}$, as well as the infinitely many Turing Machines built around $M_{M_\mathcal{F}}$ in Corollary 2, is *not* a malware but no proof exists within the formal system $\mathcal{F}$ *if* it is consistent.

Corollary 3. *For the Turing Machine $M_{M_\mathcal{F}}$ it holds that it is not a malware if and only if $\mathcal{F}$ is consistent.*

Proof. The proof is, essentially, the same as the proof of Corollary 1, since $M_{M_\mathcal{F}}$ contains $M_\mathcal{F}$ and it is constructed in such a way so that it is not a malware if and only if $M_\mathcal{F}$ does not halt on any particular input j. □

5 Discussion and directions for further research

In this paper we addressed the problem of whether it is possible to have a *complete*, in principle, classification of *all* programs either as *malware* or *non-malware*, using suitable formal systems and their proof mechanisms.

Based on Cohen's pioneering work, we showed that no algorithm exists that can classify all programs, in general, as malware or non-malware, i.e. the malware identification problem is undecidable. Further to this result, we showed that to each *formal system*, there corresponds a recursively enumerable, infinite, set of Turing Machines, which depends on the formal system's details, for which *no proof* exists, in the formal system, with respect to whether they are malware or non-malware.

From Theorem 5 and Corollary 2 it follows that, in principle, there is an infinity of programs for which a formal classification with respect to whether they are malware or not is impossible, no matter what formal system is used for this classification. This implies, that an infinite set of programs exists, which can be potentially malware, which cannot be proved to be so in any formal system we may ever devise, no matter how expressive and powerful is. Moreover, the members of this set are recursively enumerable, i.e. there exists a systematic way to list them. This, however, may provide the means to malicious parties to generate programs whose malware status is undecidable, by purely formal means. It only suffices to know the details of the formal system deployed to classify programs as malware and non-malware.

Additionally, as Corollary 3 shows, all these programs are, actually, *non-malware* programs, unless the formal system, $\mathcal{F}$, deployed for the classification task is *inconsistent*. Thus, although these programs are harmless, it is, nevertheless, impossible to classify them as such within any *consistent* formal system $\mathcal{F}$. Furthermore, if $\mathcal{F}$ *is*, actually, inconsistent, it *is* possible that the Turing Machines constructed in Corollary 2, *are* malware programs, in view of Corollaries 1 and 3. Thus, the agonizing dilemma may be the following: *Is the program under scrutiny really, a non-malware program, as guaranteed by Corollary 3 or is it true that the formal system deployed to classify the programs, as malware or non-malware inconsistent, in which case the guarantee of Corollary 3 is not valid?* We should stress the fact the proving that a given formal system $\mathcal{F}$ is consistent is a notoriously difficult problem. There exist examples of formal systems proposed in the past (e.g. *ML*, proposed by Quine in [8]) that were, later, proved (perhaps unexpectedly) to be inconsistent (*ML* was proved inconsistent by Rosser) as well as formal systems extensively used today in mathematics, whose consistency status is, *still*, unknown, such as Zermelo-Fraenkel's set theory with the axiom of choice.

As a next step, the status of the *Malware Detection Problem* can be pursued under other plausible definitions of malware behaviour either targeting, for instance, the behavior (i.e. *sequences* of specific computational

steps) or, even, the *languages* that malicious malware Turing Machines may accept (this approach is pursued in [7]). We believe that the investigation of the decidability status of any *entity* recognition problem (such as malware), can be, considerably, benefitted from a formal definition of the entities' characteristic behavior using a computational formalism, such as Turing Machines. Thus, the rich results of computability and computational complexity theory can lead to the derivation of interesting findings with respect to the fundamental difficulty of detecting such entities. Hopefully, our work is one step towards this direction.

We close our paper with the abstract of Ken Thomson's excellent Turing Award lecture (see [11]) that summarizes so succinctly our conclusions, i.e. that no automated solution can be relied on for a *complete* characterization of all programs as malware or non-malware: *To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.*

Acknowledgements
We would like to thank the anonymous reviewers for their constructive and inspiring comments.

References

1. F. Cohen. *Computer Viruses*. PhD thesis, University of Southern California, 1985.
2. F. Cohen. Computer Viruses: Theory and Experiments. *Computers & Security*, Volume 6, Issue 1, pp. 22–35, 1987.
3. M. Davis. *The Universal Computer: The Road from Leibniz to Turing*. CRC Press, 3rd edition, 2018.
4. D. Evans. *Introduction to Computing: Explorations in Language, Logic, and Machines*. CreateSpace Independent Publishing Platform, 2011.
5. J. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley series in Computer Science, 1979.
6. S.K. Kleene, *On notation for ordinal numbers*, J. Symb. Log. **3** (1938), 150–155.
7. V. Liagkou, P.E. Nastou, P. Spirakis, Y.C. Stamatiou. *On the theoretical impossibility of Panopticon Detection*. 2021, Submitted.
8. W.V. Quine. *Mahematical Logic*. Harvard University Press, Cambridge, MA, 1940.
9. P. Raattkainen. *On the Philosophical Relevance of Gödel's Incompleteness Theorems*. Revue internationale de philosophie, vol. 234, no. 4, pp. 513–534, 2005.
10. H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.
11. K. Thompson. *Reflections on trusting trust*. Commun. ACM 27, **8** (Aug 1984), 761–763. DOI:https://doi.org/10.1145/358198.358210
12. A.M. Turing. On Computable Numbers, with an Application to the Entscheidungsprobem. *Proceedings of the London Mathematical Society*, 2:230–265, 1936–7.