Proceedings of the 53rd Hawaii International Conference on System Sciences | 2020

Toward a Mathematical Understanding of the Malware Problem

Michael Stephen Fiske
Aemea Institute
mf@aemea.org

Abstract

Malware plays a significant role in breaching computer systems. Previous research has focused on malware detection even though detection is up against theoretical limits in computer science and current methods are inadequate in practice. We explain the susceptibility of computation to malware as a consequence of the instability of Turing and register machine computation. The behavior of a register machine program can be sabotaged, by making a very small change to the original, uninfected program. Stability has been studied extensively in dynamical systems and in engineering fields such as aerospace. Our primary contribution introduces mathematical tools from topology and dynamical systems to explain why register machine computation is susceptible to malware sabotage. A correspondence is constructed such that one computational step of a Turing machine maps to one iteration of a dynamical system in the x-y plane and vice versa. Using this correspondence, another contribution defines and demonstrates a structural instability in a Universal Turing machine encoding. One research direction proposes to better understand instability in conventional computation by studying non-isolated metrics on the space of Turing machines; another suggests searching for stable computation in unconventional machines.

1. Introduction

Malware can exploit a weakness in current computer systems: user authentication does not protect the execution of the user's intended action. Malware can circumvent strong authentication on a hardware token [1], even when it is tightly integrated with strong cryptographic protocols. As aptly stated by Shamir [2], "cryptography is typically bypassed, not penetrated".

It seems unlikely that malware detection methods [3, 4, 5] can solely provide an adequate solution to the malware problem. First, it is known that there is no Turing machine algorithm that can detect all malware [6]. Second, some recent malware implementations use NP problems [7] to encrypt and hide the malware [8]. Overall, detection methods are currently up against fundamental limits in theoretical computer science [9].

Rather than continue to pursue detection, we explain a register machine's [10, 11] susceptibility to malware as a consequence of the instability of conventional computation. The instability of register machine computation enables malware to sabotage the purpose of a computer program, by making small changes to an original, uninfected program. Programming languages such as `C`, `Java`, `Lisp` and `Python` depend upon branching instructions. After a branching instruction of a register machine program has been sabotaged, even if there is a routine to check if the program is behaving properly, this friendly routine may never get executed. *The sequential execution of register machine instructions cripples the program from protecting itself.*

To the best of the author's knowledge, prior research [12, 13, 14] has not attempted to understand malware susceptibility in terms of structural stability. Dynamical systems [15, 16, 17] has extensively studied stability. Our primary contribution introduces mathematical tools from topology and dynamical systems theory to explain why register machine computation is susceptible to malware sabotage. We construct a correspondence such that each computational step of a Turing machine program corresponds to one iteration of a dynamical system in the x-y plane. Using this correspondence, another contribution defines and demonstrates a structural instability in a Universal Turing machine encoding. This is relevant in practice because a programming language cannot express universal computation unless its compiler or interpreter "acts as a Universal Turing machine."

2. Unstable Computation

Two examples illustrate what we mean by *unstable computation*. Our first example is a `C` source code [18]

URI: https://hdl.handle.net/10125/64563
978-0-9981331-3-3

listing that sorts four integers. This C program shows how even small changes to a single machine instruction can substantially alter the program's behavior.

```
#include  <stdio.h>     #include  <stdlib.h>
#include  <string.h>

#define   NUM_BITS   16
int pow_2[NUM_BITS] = {0x8000, 0x4000, 0x2000,
                       0x1000, 0x800, 0x400,
                       0x200, 0x100, 0x80, 0x40,
                       0x20, 0x10, 0x8, 0x4, 0x2, 0x1};

int greater_than(int p1, int p2)
{ return (p1 > p2); }

int less_than(int p1, int p2)
{ return (p1 < p2); }

void slow_sort(int* v, int n, int (*op)(int, int)) {
   int i, k, x;
   for(i = 0; i < n; i++)
   for(k = 0; k < i; k++)   {
      if ( op(v[i], v[k]) )   {
         x = v[i];
         v[i] = v[k];
         v[k] = x;
      }
   }
}

void display_numbers(int* v, int n)   {
   int k;
   printf("\n");
   for(k = 0; k < n; k++)
      printf("%d  ", v[k] );
}

void print_binary(unsigned int v)   {
   int k;
   for(k = 0; k < NUM_BITS; k++)  {

      if (v / pow_2[k])  printf("1 ");
      else               printf("0 ");

      v %= pow_2[k];
   }
   printf("\n");
}

void sort_pr(int* nums, int n, char* fn,
             int (*op)(int, int) )        {
  slow_sort(nums, n, op);
  display_numbers(nums, n);
  printf("   address of instruction");
  printf("%s  \n          ", fn);
  print_binary((unsigned int) op);
}

int main(int argc, char* argv[])
{
   int nums[4] = {6, 9, 7, 8};
   display_numbers(nums, 4);
   printf("\n");
   sort_pr(nums, 4, "less_than", less_than);
   sort_pr(nums, 4, "greater_than", greater_than);
   return 0;
}
```

Figure 1 shows an execution of this C program. The ordering of the sorted numbers $(6, 7, 8, 9)$ is reversed to $(9, 8, 7, 6)$, by flipping only two bits of one instruction. This C program exhibits *unstable computation* because a small change (i.e., flipping two bits) in the C program causes a substantial change to the outcome of its computation: namely, it changes from sorting integers in

```
~MacBook-Air:sort$  ./sort
6 9 7 8

6 7 8 9  address of instruction less_than
         1 1 0 1 1 0 1 0 0 1 0 0 0 0 0 0

9 8 7 6  address of instruction greater_than
         1 1 0 1 1 0 1 0 0 0 0 1 0 0 0 0
```

Figure 1. Sorting order reversal

ascending order to sorting integers in descending order.

In Figure 2, the code demonstrates how to hijack a register machine program, by inserting only one `jmp WVCTF` instruction or changing the address of one legitimate `jmp` instruction to `WVCTF`.

```
WVCTF:     mov   eax,   drl
           jmp   Loc1
Loc2:      mov   edi,   [eax]
LOWVCTF:   pop   ecx
           jecxz SFMM
           inc   eax
           mov   esi,   ecx
           dec   eax
           nop
           mov   eax,   0d601h
           jmp   Loc3
Loc1:      mov   ebx,   [eax+10h]
           jmp   Loc2
Loc3:      pop   edx
           pop   ecx
           nop
           call  edi
           jmp   LOWVCTF
SFMM:      pop   ebx
           pop   eax
           stc
```

Figure 2. Polymorphic malware instructions

3. Motivating Stable Computation

Register machines execute one instruction at a time. After a register machine program has been hijacked, even if there is a routine to check if the program is behaving properly, *this friendly routine may never get executed. The sequential execution of register machine instructions cripples the program from protecting itself against malware*. Typical programming languages (e.g., C, Java, Lisp and Python) are Turing complete and depend upon branching instructions. While conditional branching instructions are not required for universal computation, Rojas's methods [19] still use unconditional branching and program self-modification. Moreover, about 75% to 80% of the control flow instructions, executed on register machines, are conditional branch instructions. (See figure A.14 in [11].)

These observations suggest that a computer program's purpose can be subverted because the register machine behavior is not always stable when small changes are made to its program. Why is this insight useful for designing malware resistant computation?

Overall, we seek malware resistant computation based on the following principle: design the computation so that if malware makes a small change, the program's purpose is stable; if a larger change is made, the program's purpose is completely destroyed. Our goal is to create stable computation that is incomprehensible to malware authors [20] so that it is far more challenging for malware to subvert the program's behavior without completely destroying it.

4. A Program is a Dynamical System

In dynamical systems, stability has been studied for over 80 years [21]. There is currently no mathematical definition of stable computation. Thus, in this section, our primary goal is to explain how the execution of a computer program corresponds to the iteration of a dynamical system in the x-y plane. This correspondence should enable us to apply powerful, mathematical tools to computation, that already have been developed in dynamical systems theory.

One of our goals is to reach some topological insight on understanding the instability in Turing and register machine computation. We seek topological insight because *open sets* in a *topology* provide a more general and flexible method for characterizing *closeness* than Euclidian distance. This is ideally how we would like to model *small changes in a computer program*.

With a simple example shown in Figure 3, we informally describe how each Turing machine (TM) [22] corresponds to a dynamical system, generated from a finite set of affine functions in the x-y plane. The appendix provides a comprehensive, mathematical treatment of this correspondence. The appendix shows that any computer program can be mapped to a corresponding dynamical system. Furthermore, one iteration of this dynamical system computes a result that is equivalent to the execution of one computational step of the Turing machine.

It is well-known that register machine computation is equivalent to TM computation and both formalisms model the behavior of a digital computer. (In [10], see footnote 19, page 386 and chapter 5.) This equivalence justifies our constructing a map from a Turing machine to a dynamical system in the x-y plane and concluding that these dynamical systems can characterize instability in register machine programs.

First, a TM is reviewed. A TM has a *tape* T that is represented as a function $T : \mathbb{Z} \to A$ where $\mathbb{Z}$ is the integers and $A = \{a_1, \ldots, a_n\}$ is a finite set of *alphabet symbols* that are read from and written to tape squares on the tape. Here $T_i = T(i)$ is the alphabet symbol on tape square i. There is a finite set of *machine states* $Q = \{q_1, \ldots, q_m\}$ and a distinct halting state h. The function $\eta : Q \times A \to Q \cup \{h\} \times A \times \{-1, +1\}$ specifies the *program instructions*. The execution of one instruction is called a *computational step* of the TM.

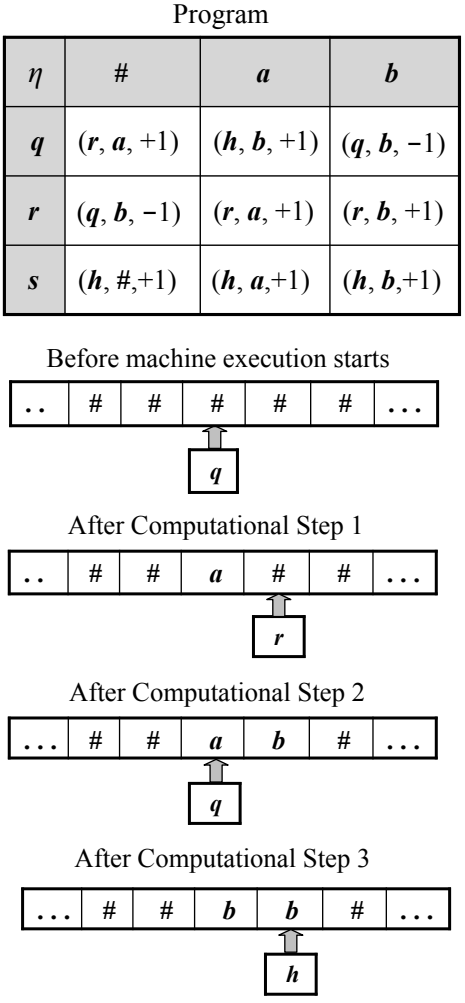


Figure 3. Turing machine execution steps

Figure 3 shows the execution of a simple TM with alphabet $A = \{\#, a, b\}$ and states $Q = \{q, r, s\}$. The initial state is q. Initially, the tape has a $\#$ (blank) symbol in every tape square. The first instruction executed is $\eta(q, \#) = (r, a, +1)$: it replaces the $\#$ with an a on the tape, jumps to state r, and then the tape head moves right one tape square.

4.1. The ϕ Correspondence

Set base $B = |A| + |Q| + 1$. Define value function $\nu : \{h\} \cup Q \cup A \to \{0, 1, \ldots, B-1\}$ as $\nu(h) = 0$, $\nu(a_i) = i$ and $\nu(q_i) = i + |A|$. ν maps each alphabet symbol and each state to a unique symbol in base B.

ϕ is the map that creates a one-to-one correspondence from the program instructions to a finite set of affine functions in the x-y plane. ϕ maps each machine configuration to a unique point in the x-y plane. ϕ maps right instruction $\eta(q, T_k) = (r, \alpha, +1)$ to affine function $f(x, y) = (Bx + m, \frac{1}{B}y + n)$, where $m = -B^2\nu(T_k)$ and $n = B\nu(r) + \nu(\alpha) - \nu(q)$. ϕ maps left instruction $\eta(q, T_k) = (r, \alpha, -1)$ to affine function $g(x, y) = (\frac{1}{B}x + m, By + n)$, where $m = B\nu(T_{k-1}) + \nu(\alpha) - \nu(T_k)$ and $n = B\nu(r) - B^2\nu(q) - B\nu(T_{k-1})$.

After each step, the TM is in some *configuration* $(q, k, T) \in Q \times \mathbb{Z} \times A^{\mathbb{Z}}$. ϕ maps (q, k, T) to $\big(\sum_{j=-1}^{\infty} \nu(T_{k+j+1})B^{-j},\ B\nu(q) + \sum_{j=0}^{\infty} \nu(T_{k-j-1})B^{-j}\big)$ in the x-y plane. Both the x and y coordinates are convergent sums. Since the tape contains a finite number of non-blank symbols, after a finite number of terms, the tail of each sum is a geometric series.

4.2. One Computational Step is One Iteration

The execution of the TM in Figure 3 is faithfully modelled by iterating the corresponding affine functions. Since alphabet $A = \{\#, a, b\}$ and states $Q = \{q, r, s\}$, base $B = 7$. Also, $\nu(h) = 0$, $\nu(\#) = 1$, $\nu(a) = 2$, $\nu(b) = 3$, $\nu(q) = 4$, $\nu(r) = 5$ and $\nu(s) = 6$.

ϕ maps the initial machine configuration in Figure 3 to point $p = (p_x, p_y)$ in the x-y plane, where $p_x = B\nu(\#) + \frac{\nu(\#)}{1-\frac{1}{7}} = 8\frac{1}{6}$ and $p_y = B\nu(q) + \frac{\nu(\#)}{1-\frac{1}{7}} = 29\frac{1}{6}$.

The first step executes instruction $\eta(q, \#) = (r, a, +1)$, which corresponds to applying affine function $f_1(x, y) = (7x - 49, \frac{1}{7}y + 33)$ to p. For f_1, $m = -7^2\nu(\#) = -49$ and $n = 7\nu(r) + \nu(a) - \nu(q) = 33$. The first iteration is $f_1(8\frac{1}{6}, 29\frac{1}{6}) = (8\frac{1}{6}, 37\frac{1}{6})$.

The second step executes $\eta(r, \#) = (q, b, -1)$. The second step corresponds to applying affine function $f_2(x, y) = (\frac{1}{7}x + 16, 7y - 231)$ to point $(8\frac{1}{6}, 37\frac{1}{6})$, where $m = 7\nu(a) + \nu(b) - \nu(\#) = 16$ and $n = 7\nu(q) - 7^2\nu(r) - 7\nu(a) = -231$. The result of the second iteration is $f_2(8\frac{1}{6}, 37\frac{1}{6}) = (17\frac{1}{6}, 29\frac{1}{6})$. The third iteration is computed similarly. The *orbit* of p includes p and all points reached after each iteration.

As a summary, executing a Turing machine corresponds to iterating a discrete, autonomous system in the x-y plane; the dynamical system consists of a finite number of affine functions, whose domains lie in distinct unit squares. If configuration (q, k, T) halts after n computational steps, then the orbit of $\phi(q, k, T)$ exits one of the unit squares on the nth iteration, and enters the halting attractor. (See definition 10 in the appendix.) If configuration (r, j, S) never halts, then the orbit of $\phi(r, j, S)$ remains in these unit squares forever.

5. Applying Topology to Computation

In this section, we work toward formalizing our intuitive explanation of unstable computation discussed in the prior sections. Structural stability characterizes the stability of a dynamical system under small changes or perturbations [23]. With this in mind, we review some mathematical definitions and italicize their names.

A *topology* [24] on a set X is a collection $\mathcal{T}$ of subsets of X having the following properties: (a) $\emptyset$ and X are both in $\mathcal{T}$; (b) The union of the elements of any subcollection of $\mathcal{T}$ is in $\mathcal{T}$; (c) The intersection of the elements of any finite subcollection of $\mathcal{T}$ is in $\mathcal{T}$. A set X for which a topology $\mathcal{T}$ has been specified is called a *topological space*. A subset U of X is called *open* in this topology if U belongs to the collection $\mathcal{T}$.

Let $\mathbb{R}$ be the real numbers. As an example of a topological space, for real numbers a and b, the open intervals $(a, b) = \{x \in \mathbb{R} : a < x < b\}$ form a basis for the standard topology on the real numbers, generated from arbitrary unions of open intervals and finite intersections of open intervals.

Let X and Y be topological spaces. The function $f : X \to Y$ is *continuous* if for any open subset U of Y, the inverse image $f^{-1}(U) = \{x \in X : f(x) \text{ lies in } U\}$ is open in X's topology. The function $h : X \to Y$ is a *homeomorphism* if h is continuous, h is one-to-one and onto, and h's inverse $h^{-1} : Y \to X$ is continuous.

5.1. Topological Conjugacy

A *discrete, dynamical system* is a function $f : X \to X$, where X is a topological space. The *orbit* of p is $\{f^n(p) : n \in \mathbb{N}\}$, which is the set of points, obtained by iterating f on initial point p. Consider discrete, dynamical systems $f : X \to X$ and $g : Y \to Y$. f and g are *topologically conjugate* if f and g are continuous functions and there exists a homeomorphism $h : X \to Y$ such that $h \circ f = g \circ h$. Topologically conjugate functions exhibit equivalent dynamics. For example, if f is topologically conjugate to g via h and p is a fixed point for f, then $h(p)$ is a fixed point for g. Similarly, h induces a one-to-one correspondence between the periodic points of f and g. Suppose $p \in X$ has period n with respect to f. Then $f^n(p) = p$, so $h(p) = h(f^n(p)) = h \circ f(f^{n-1}(p)) = g \circ h(f^{n-1}(p)) =$

$g^2 \circ h(f^{n-2}(p)) = \cdots = g^{n-1} \circ h(f(p)) = g^n(h(p))$. Thus, $h(p)$ is a periodic point of g with period n.

Topological conjugacy is a useful notion for computation because after a Turing machine has halted, its halted machine configuration represents what the Turing machine has computed. Furthermore, each halted machine configuration corresponds to a fixed point (halting point) of the dynamical system. If h is a topological conjugacy with $h \circ f = g \circ h$, then p is a fixed point of f if and only if $h(p)$ is a fixed point. Hence, a topological conjugacy between machines $\mathcal{M}_1$ and $\mathcal{M}_2$ induces a one-to-one correspondence between the halting configurations of $\mathcal{M}_1$ and $\mathcal{M}_2$.

In the appendix, the ϕ correspondence, between halting configurations of a Turing machine and fixed points of the corresponding dynamical system in the x-y plane, can be applied to any Turing machine. (See definitions 10, 11, and theorem 2.) Thus, for any computer program $\mathcal{P}$, regardless of $\mathcal{P}$'s complexity or size, there exists a ϕ correspondence that is general enough to apply to program $\mathcal{P}$.

Lastly, there is an algorithm $\mathcal{A}_\phi$ that receives program $\mathcal{P}$, initial tape T, and initial state q as input. During the computation, $\mathcal{A}_\phi$ prints the initial point p in the x-y plane and a finite set of affine functions that represents the dynamical system, corresponding to $\mathcal{P}$. The computations performed by algorithm $\mathcal{A}_\phi$ are explicitly specified in definition 6, remark 1, definition 8, and definition 9. Mathematical proofs – that the computations performed by $\mathcal{A}_\phi$ are correct – are provided in lemma 2, lemma 3, and theorem 2.

5.2. Metric Spaces and Structural Stability

A *metric space* is a set X and a function (metric) $d : X \times X \to \mathbb{R}$, such that the following three conditions hold. (i) $d(a, b) \geq 0$ for all $a, b \in X$ where equality holds if and only if $a = b$. (ii) $d(a, b) = d(b, a)$ for all $a, b \in X$. (Symmetric). (iii) $d(a, b) \leq d(a, c) + d(c, b)$ for all $a, b, c \in X$. (Triangle inequality).

Given $\epsilon > 0$, define the ϵ-ball $B_d(x, \epsilon) = \{y \in X : d(x, y) < \epsilon\}$. The collection of all ϵ-balls, where $\epsilon > 0$ and $x \in X$, is a basis for a metric topology on X, induced by d. In the appendix, lemma 1 implies that for a fixed Turing machine, the set of all its machine configurations is a metric space. This means metric ρ, defined in lemma 1, on the set of all machine configurations can measure the closeness of two machines.

Let (X, d) be a metric space. The C^0 distance between functions $f : X \to X$ and $g : X \to X$ is given by $\rho_0(f, g) = sup\{d\big(f(x), g(x)\big) : x \in X\}$, where sup is the least upper bound. A function $f : X \to X$ is said to be C^0 *structurally stable* on X if there exists $\epsilon > 0$ such that whenever $\rho_0(f, g) < \epsilon$ for $g : X \to X$, then f is topologically conjugate to g.

In other words, a dynamical system f is structurally stable if for all dynamical systems g that are close to f, then f is topologically conjugate to g. Structural stability is a mathematical tool that we want to apply to computer programs (i.e., Turing machines). We can accomplish this by using the ϕ correspondence between the execution of a computer program and the iteration of the program's corresponding dynamical system.

6. Instability in Turing Computation

In this section, we apply structural stability to a particular Universal Turing machine (UTM), which hereafter will be called machine $\mathcal{U}$. We show that $\mathcal{U}$'s computation is unstable under arbitrarily small changes.

Our demonstration of $\mathcal{U}$'s unstable computation is relevant to practical applications based on the following line of thinking. A C compiler acts as a Universal Turing machine, where the programs executed are valid C programs. More concretely, a valid C compiler can be executed with a finite table of Turing instructions, called $\mathcal{P}_c$, according to the encoding used by universal machine $\mathcal{U}$. Since $\mathcal{U}$'s computation is unstable, there could be a distinct, finite table of Turing instructions, called $\mathcal{P}_{mal}$, that subverts the purpose of valid C compiler $\mathcal{P}_c$, and $\mathcal{P}_{mal}$ is very close to $\mathcal{P}_c$.

6.1. An Unstable UTM Encoding

The encoding used by machine $\mathcal{U}$ is shown in Figure 4 and Figure 5. Machine $\mathcal{U}$ has alphabet $\{\#, 0, 1\}$, and its states are $Q = \{q_1, q_2, \ldots, q_U\}$. A program table for $\mathcal{U}$ is not provided, but a table can be constructed, based on the description provided here.

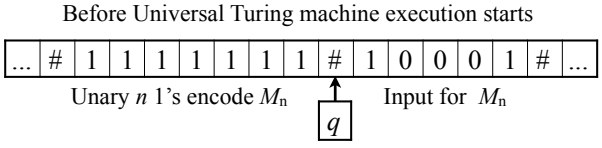


Figure 4. Universal Turing Machine $\mathcal{U}$

We assume $\mathcal{U}$ starts execution, scanning a blank symbol #. A sequence of n 1's to the left of the $\mathcal{U}$'s initial tape head location is a unary encoding for the nth Turing machine M_n that machine $\mathcal{U}$ executes. The sequence of 0's and 1's to the right of $\mathcal{U}$'s initial tape head location are the input to machine M_n. Also, it is well-known that a finite number of states and alphabet $\{\#, 0, 1\}$ can represent any TM [25].

M_1

η	#	0	1
q_1	(h, 0,−1)	(h, 0,−1)	(h, 0,−1)

M_{215}

η	#	0	1
q_1	(h, 0,+1)	(q_1, #,+1)	(h, #,+1)

M_{1728}

η	#	0	1
q_1	(q_1, #,+1)	(q_1, #,+1)	(q_1, #,+1)

M_{1729}

η	#	0	1
q_1	(h, 0,−1)	(h, 0,−1)	(h, 0,−1)
q_2	(h, 0,−1)	(h, 0,−1)	(h, 0,−1)

$M_{12^3+18^6+1}$

η	#	0	1
q_1	(h, 0,−1)	(h, 0,−1)	(h, 0,−1)
q_2	(h, 0,−1)	(h, 0,−1)	(h, 0,−1)
q_3	(h, 0,−1)	(h, 0,−1)	(h, 0,−1)

Figure 5. Some machines executed by $\mathcal{U}$

The enumeration of each Turing machine by machine $\mathcal{U}$ works as follows: The number of distinct programs (η's) for a fixed number of states $|Q|$ is $\big(2|A||Q| + 2|A|\big)^{|A||Q|}$. The first 12^3 machines are all the possible Turing programs with $Q = \{q_1\}$ and $A = \{\#, 0, 1\}$. Figure 5 shows the program for M_1 that $\mathcal{U}$ explicitly constructs for the unary representation of 1 before $\mathcal{U}$ starts executing M_1; the input for M_1 is to the right of $\mathcal{U}$'s initial tape head location. When the unary value is $12^3 + 18^6 + 1$, then all possible Turing programs with $Q_1 = \{q_1\}$ and $Q_2 = \{q_1, q_2\}$ have been exhausted, so the enumeration updates the state set to $Q_3 = \{q_1, q_2, q_3\}$. The program for $M_{12^3+18^6+1}$ is also shown in Figure 5. This encoding works for each finite set of states $Q_n = \{q_1, \ldots, q_n\}$ for every $n \in \mathbb{N}$.

Based on the ϕ correspondence, our next definition measures a distance between two Turing machines in terms of the encoding used by machine $\mathcal{U}$. Note that our metric definition depends upon the encoding that machine $\mathcal{U}$ uses.

Definition 1. *Turing Machine Encoding Metric ρ*
For universal machine $\mathcal{U}$, let M_n and M_m denote the TMs encoded by n and m 1's, respectively. Set $\nu(h) = 0$, $\nu(\#) = 1$, $\nu(0) = 2$, $\nu(1) = 3$, $\nu(q_1) = 4$ $\ldots$ $\nu(q_U) = U + 3$ where the states of the UTM are $q_1, \ldots q_U$. Set $B = U + 4$. For $m \leq n$, define $\rho(M_n, M_m) = \Big|\sum_{j=0}^{n-1} \nu(1)B^{-j} + \sum_{j=n}^{\infty} \nu(\#)B^{-j} - \sum_{j=0}^{m-1} \nu(1)B^{-j} - \sum_{j=m}^{\infty} \nu(\#)B^{-j}\Big| = 2\sum_{j=m}^{n-1} B^{-j}$

Theorem 1. *Universal Turing machine $\mathcal{U}$ has unstable computation in the following sense. For any $\epsilon > 0$, there exist two distinct Turing machines closer than ϵ with respect to ρ such that the respective dynamical systems of these two machines are not topologically conjugate.*

This means two different Turing machines that are arbitrarily close with respect to ρ exhibit different computing behavior.

PROOF. As above, let $Q_k = \{q_1, q_2, \ldots, q_k\}$ and thus, $|Q_k| = k$. Derived from $\sum_{k=1}^{n}(2|A|k + 2|A|)^{|A|k}$, define function $f : \mathbb{N} \to \mathbb{N}$ where $f(n) = \sum_{k=1}^{n}(6k + 6)^{3k} + 1$. Then the machine $M_{f(n)}$ has only halting configurations for all n. Machine $M_{\nu(n)}$ is shown in Figure 6. For each interval of natural numbers $[f(n), f(n+1)]$, $M_{\nu(n)}$ satisfies $f(n) < \nu(n) < f(n+1)$. Also, $M_{\nu(n)}$ has no halting configurations. Thus, $M_{\nu(n)}$ is not topologically conjugate to $M_{f(n)}$ for all n. Further, $\rho(M_{f(n)}, M_{\nu(n)}) < \frac{2B}{B-1}B^{-f(n)}$, implies that for all $\epsilon > 0$, there are two machines closer than ϵ and they are not topologically conjugate. □

η	#	0	1
q_1	(q_2, 0,+1)	(q_2, 1,−1)	(q_2, #,−1)
$\ldots$			
q_k	(q_{k+1}, 0,+1)	(q_{k+1}, 1,−1)	(q_{k+1}, #,−1)
$\ldots$			
q_n	(q_1, 0,+1)	(q_1, 1,−1)	(q_1, #,−1)

Figure 6. Turing Machine $M_{\nu(n)}$

6.2. Non-Isolated Metrics are Useful

Let X be a topological space. A *separation* of X is a pair U, V of disjoint open subsets of X whose union is X. X is *connected* if there does not exist a separation of X. A topological space is *totally disconnected* if its only connected subsets are one-point sets.

One of our goals is to mathematically model the stability of computation with respect to small changes.

If metric D on topological space X satisfies the property that for all $x \in X$, there exists an $\epsilon > 0$ such that $B_D(\epsilon, x)$ only contains x, then D is called an *isolated metric*. An isolated metric cannot quantify an arbitrarily small change to a machine, so it has no utility for defining stable computation.

Let $M_1, M_2, \ldots$ be an enumeration of all Turing machines for some UTM. For each m, define the metric $\gamma(M_m, M_m) = 0$ and $\gamma(M_m, M_n) = 1$ when $m \neq n$. γ vacuously satisfies the stability property since any two distinct Turing machines are a distance of 1 apart. γ is an example of an isolated metric. Moreover, for every Universal Turing machine enumeration $\{M_1, M_2, \ldots\}$, the machine space $\mathfrak{M} = \{M_m : m \in \mathbb{N}\}$, induced by γ, is totally disconnected.

A metric is *non-isolated* on machine space $\mathfrak{M}$ if the metric does not disconnect $\mathfrak{M}$. A longer-term goal is to mathematically characterize non-isolated metrics and use them to search for machines M in $\mathfrak{M}$ that exhibit stable computation in an open neighborhood of M.

Determining whether a metric is isolated, independent of the UTM, seems subtle. This is shown in example 1: a metric r is defined to measure how closely two orbits of two Turing machines match, which seems to be a natural measure of closeness.

Example 1.

Let $M_1, M_2, \ldots$ be an enumeration of all Turing machines for some UTM. Let η_n be the program instructions for machine M_n and η_m the program instructions for M_m. $I = (q_0, i, T)$ is an initial machine configuration, where $1 \leq i \leq n$ and $T(j) = a_j \in \{0, 1, \#\}$ for $1 \leq j \leq n$ and $T(j) = \#$ when $j \leq 0$ or $j > n$. Let $(q_{(1,n)}, a_{(1,n)}) \ldots (q_{(k,n)}, a_{(k,n)}), (q_{(k+1,n)}, a_{(k+1,n)})$ be the sequence of inputs for η_n in the first k computational steps of M_n and $(q_{(1,m)}, a_{(1,m)}) \ldots (q_{(k,m)}, a_{(k,m)}), (q_{(k+1,m)}, a_{(k+1,m)})$ be the sequence of inputs for η_m. Assume both machines have the same initial starting state $q_0 = q_{(1,n)} = q_{(1,m)}$. Let $k(I, M_n, M_m)$ be the number of computational steps until M_n and M_m differ with respect to their execution steps, i.e. the first k such that $\eta_n(q_{(k,n)}, a_{(k,n)}) \neq \eta_m(q_{(k,m)}, a_{(k,m)})$.

For only configuration I, the distance between machines M_m and M_n is defined as $\mu(I, M_n, M_m) = 2^{-k+1}$. In the special cases, where $\eta_n(q_{(k,n)}, a_{(k,n)}) = \eta_m(q_{(k,m)}, a_{(k,m)})$ for all k either because it is an immortal orbit or a halting orbit, then $\mu(I, M_n, M_m) = 0$. Let Γ be the set of all tapes T such that $T(j) \in \{0, 1, \#\}$ for $1 \leq j \leq n$ and $T(j) = \#$ when $j \leq 0$ or $j > n$. Define $r(M_m, M_n) = \sup\{\mu(I, M_n, M_m) : I \in Q \times \{1, \ldots, n\} \times \Gamma\}$.

If $m \neq n$, there is some instruction where $\eta_m(q, a) \neq \eta_n(q, a)$. Consider initial configuration $I = (q, 1, T)$ where $T(1) = a$. Then $\mu(I, M_n, M_m) = 2^{-1+1}$, so $r(M_m, M_n) = 1$. Further, $r(M_m, M_n) = 0$ when $m = n$.

Although metric r seems to carefully measure the distance between machines based on how many computational steps match, r is the same metric as γ, so r is an isolated metric.

7. Two Research Directions

Classical dynamical systems theory helps to introduce new notions for characterizing the stability of digital computer programs. There is empirical evidence that malware can change the purpose of a computer program by making very small changes to the original program; only one address of one branch instruction in a digital computer program needs to be changed in order to subvert the machine to execute the malware. Malware authors are able to exploit this vulnerability because register machines rely on branch instructions and execute their instructions one at a time.

In section 6, we constructed a universal machine $\mathcal{U}$ that is unstable with respect to metric ρ, Further research should study other non-isolated metrics and their relationship to other Universal Turing machine encodings. A goal is to find more general mathematical conditions when a TM computation is structurally unstable that includes a search for non-isolated metrics on the space of Turing machines. If there exist conditions for stable TM computation, then this know-how could help design more robust digital computer programs.

Using our mathematical tools for understanding stable computation, a second research direction should further develop machines that can simultaneously execute multiple instructions. Using the active element machine's inherent parallelism [26], it might be possible to program redundancy in the `element` and `connection` commands, and to repair a sabotaged program with `meta` commands. A goal is to build active element machine programs whose purpose does not change even when some of the commands in a program have been sabotaged. Overall, we propose to search for unconventional machines that can execute stable computation.

Acknowledgments

I am deeply grateful to Lawrence Reeves, President of AFCEA Monterey Bay chapter, for his introduction to NPS faculty, and sage advice. I am deeply grateful to Dan Boger and Peter Denning for a meeting in 2013. I am deeply grateful to the reviewers for their comments.

References

[1] Keith Mayes and Konstantinos Markantonakis (editors). Smart Cards, Tokens, Security and Applications. Springer, 2008.

[2] Adi Shamir. Cryptography: State of the Science. ACM. Alan M. Turing Award Lecture. June 8, 2003.

[3] John Mitchell and Elizabeth Stillson. Detection of Malicious Programs. U.S. Patent 7,870,610, 2011.

[4] Andreas Moser, Chris Kruegel and Engin Kirda. "Limits of Static Analysis for Malware Detection." IEEE. 23rd Annual Computer Security Applications Conf., 2007.

[5] Diego Zamboni (editor). Proc. of the 5th Intl. Conf. on Detection of Intrusions and Malware LNCS. Springer. July 2008.

[6] Fred Cohen. "Computer Viruses Theory and Experiments. Computers and Security." 6(1) 22–35, Feb. 1987.

[7] Stephen Cook. "The P versus NP Problem." Clay Math Institute, 2013.

[8] Eric Filiol. "Malicious Cryptology and Mathematics." Cryptography and Security in Computing. Chapter 2. Intech, March 7, 2012.

[9] Eric Filiol. Computer viruses: from theory to applications. Springer, 2005.

[10] Harold Abelson and Gerald J. Sussman with J. Sussman. Structure and Interpretation of Computer Programs. Second Edition, MIT Press, 1996.

[11] John Hennessy and David Patterson. Computer Architecture. 5th Edition, Morgan Kaufmann, 2012.

[12] Len Adleman. "An Abstract Theory of Computer Viruses." Advances in Cryptology – CRYPTO 2008. LNCS 403, Springer, 1988.

[13] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. "On Abstract Computer Virology from a Recursion-theoretic Perspective." Journal in Computer Virology. 1, No. 3-4, 2006.

[14] Hubert Godfroy and Jean-Yves Marion. "Abstract Self Modifying Machines." HAL CCSD, 2016.

[15] Jacob Palis and Stephen Smale. "Structural Stability Theorems." Proc. Symp. Pure Math. AMS. 14, 223–232, 1970.

[16] Clark Robinson. "Structural Stability of C^1 Diffeomorphisms." Journal of Differential Equations. 22, 28–73, 1976.

[17] Keonhee Lee and Kazuhiro Sakai. "Structural stability of vector fields with shadowing." Journal of Differential Equations. 232, 303–313, 2007.

[18] Brian Kernighan and Dennis Ritchie. The C Programming Language. 2nd Edition, Prentice Hall, 1988.

[19] Raul Rojas. "Conditional Branching is not Necessary for Universal Computation in von Neumann Computers." Journal of Universal Computer Science. 2, No. 11, 756–768, 1996.

[20] Michael S. Fiske. "Turing Incomputable Computation." Turing-100 Proceedings. Alan Turing Centenary. EasyChair 10, 69–91, 2012.

[21] Aleksandr Andronov and Lev Pontrjagin. "Systèmes Grossiers." Dokl. Akad. Nauk., SSSR, 14, 247–251, 1937.

[22] Alan M. Turing. "On computable numbers, with an application to the Entscheidungsproblem." Proc. London Math. Soc. Series 2. 42 Parts 3, 4, 230–265, 1936.

[23] Clark Robinson. Dynamical Systems. Stability, Symbolic Dynamics and Chaos. CRC Press, 1996.

[24] James R. Munkres. Topology: A First Course. Prentice-Hall, 1975.

[25] Claude Shannon. "A Universal Turing Machine with Two Internal States." Automata Studies. Annals of Mathematics Studies, No. 34, Princeton Univ. Press, 157–165, 1956.

[26] Michael S. Fiske. "The Active Element Machine." Proc. of Comp. Intelligence. Autonomous Systems. 391, 69–96, Springer, 2011.

[27] Marvin Minsky. Computation: Finite and Infinite Machines. Prentice-Hall, 1967.

Appendix

A Turing Machine is defined; function η specifies the *program instructions* instead of quintuples [27].

Definition 2. *A Turing machine is a triple* (Q, A, η)*:*

(1) Q is a finite set of states that does not contain a unique halting state h.

(2) A is a finite alphabet whose symbols are read from and written to a tape. $\#$ is the blank symbol.

(3) -1 and $+1$ represent advancing the tape head one square to the left or right, respectively.

(4) $\eta : Q \times A \to Q \cup \{h\} \times A \times \{-1, +1\}$ specifies the program instructions.

For each q in Q and α in A, $\eta(q, \alpha) = (r, \beta, x)$ describes how the machine executes one computational step. When in state q and reading alphabet symbol α on the tape, machine (Q, A, η) jumps to state r, and replaces symbol α with β on the tape.

If $x = -1$, the tape head moves one square to the left on the tape. If $x = +1$, the tape head moves one square to the right on the tape. If $r = h$, machine (Q, A, η) enters halting state h, and stops executing.

Definition 3. *Turing Machine Tape*

Function $T : \mathbb{Z} \to A$ represents the tape T. Before machine execution begins, the tape contains a finite number of non-blank symbols. The alphabet symbol on the k^{th} square of the tape is denoted as $T(k)$ or T_k.

Definition 4. *Tape Head Location*

Let (Q, A, η) be a Turing machine with tape T. A configuration is an element of the set $C = Q \cup \{h\} \times \mathbb{Z} \times \{T | T : \mathbb{Z} \to A\}$. If (q, k, T) is a configuration, then k is the tape head location on tape T.

Definition 5. *Turing Machine Computational Step*

Consider (Q, A, η) in configuration (q, k, T), where $T(k) = a$. After executing one instruction, the new configuration is determined by one of the four cases.

(1) $(r, k-1, S)$ if $\eta(q, a) = (r, b, -1)$.

(2) $(r, k+1, S)$ if $\eta(q, a) = (r, b, +1)$.

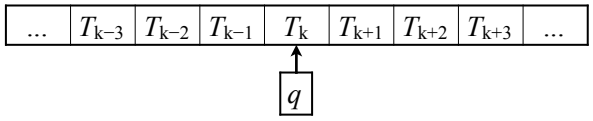
(3) $(h, k-1, S)$ if $\eta(q, a) = (h, b, -1)$.

(4) $(h, k+1, S)$ if $\eta(q, a) = (h, b, +1)$.

In cases 1–4, the new tape S satisfies $S(j) = T(j)$ when $j \neq k$ and $S(k) = b$. In cases 3 and 4, the machine execution halts after the step is completed. If the machine is currently in configuration (q_0, k_0, T_0) and over the next n steps the sequence of configurations is (q_0, k_0, T_0), (q_1, k_1, T_1), $\ldots$, (q_n, k_n, T_n), then this sequence is called the next n computational steps. (q, k, T) is *immortal* if (q, k, T) never halts.

This next part shows the correspondence between the execution of the Turing machine and the iteration of a dynamical system in the x-y plane P.

Definition 6. *Machine Configurations $\Longleftrightarrow$ Plane P*



For a fixed machine, each configuration (q, k, T) represents a unique point (x, y) in the x-y plane P. Let $\mathfrak{C}$ be the set of configurations.

Coordinate function $x : \mathfrak{C} \to P$ is $x(q, k, T) = T_k T_{k+1} \,.\, T_{k+2} T_{k+3} T_{k+4} \ldots$, where $x(q, k, T)$ is the number $B\nu(T_k) + \nu(T_{k+1}) + \sum_{j=1}^{\infty} \nu(T_{k+j+1}) B^{-j}$.

Coordinate function $y : \mathfrak{C} \to P$ is $y(q, k, T) = q\, T_{k-1} \,.\, T_{k-2} T_{k-3} T_{k-4} \ldots$, where $y(q, k, T)$ is the number $B\nu(q) + \nu(T_{k-1}) + \sum_{j=1}^{\infty} \nu(T_{k-j-1}) B^{-j}$. Define function $\phi : \mathfrak{C} \to P$ as $\phi(q, k, T) = \big(x(q, k, T), y(q, k, T)\big)$. Let $\mathfrak{N}$ be the immortal configurations in $\mathfrak{C}$.

Definition 7. *Equivalent Configurations*

$(q, k, T) \sim (r, j, V)$ if $q = r$ and $T(m) = V(m + j - k)$ for every integer m. Configurations (q, k, T) and (r, j, V) are called equivalent, since $\sim$ is an equivalence relation on $\mathfrak{C}$. $\overline{\mathfrak{C}}$ is the set of all equivalence classes $[(q, k, T)]$ on $\mathfrak{C}$. ϕ maps every configuration in equivalence class $[(q, k, T)]$ to the same point in P.

$(\mathfrak{d}, X)$ is a metric space if the three conditions hold: (1) $\mathfrak{d}(a, b) \geq 0$ for all $a, b \in X$ where equality holds if and only if $a = b$. (2) $\mathfrak{d}(a, b) = \mathfrak{d}(b, a)$ for all $a, b \in X$. (Symmetric.) (3) $\mathfrak{d}(a, b) \leq \mathfrak{d}(a, c) + \mathfrak{d}(c, b)$ for all $a, b, c \in X$. (Triangle inequality.)

Lemma 1. *$(\rho, \overline{\mathfrak{C}})$ is a metric space where ρ is induced via ϕ by the Euclidean metric in P.*

PROOF. For points $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ in plane P, let d be the Euclidean metric $d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Let $u = [(q, k, S)]$, $w = [(r, l, T)]$ be elements of $\overline{\mathfrak{C}}$. Define $\rho : \overline{\mathfrak{C}} \times \overline{\mathfrak{C}} \to \mathbb{R}$ as $\rho(u, w) = d\big(\phi(q, k, S), \phi(r, l, T)\big) = \sqrt{\big(x(q, k, S) - x(r, l, T)\big)^2 + \big(y(q, k, S) - y(r, l, T)\big)^2}$.

The symmetric property and the triangle inequality hold for ρ because d is a metric. In regard to property 1, $\rho(u, w) \geq 0$ because d is a metric.

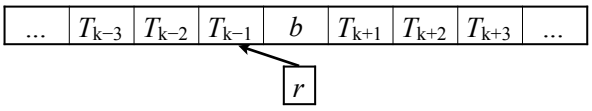
The condition $\rho(u, w) = 0$ if and only $u = w$ also holds. If $u = w$, then $(q, k, S) \sim (r, l, T)$. This implies $q = r$ and $S(m) = T(m + l - k)$ for every integer m, so $\phi(q, k, S) = \phi(r, l, T)$. Since d is a metric, $\rho(u, w) = 0$. If $u \neq w$, then $\phi(q, k, S) \neq \phi(r, l, T)$, which implies $\rho(u, w) = d\big(\phi(q, k, S), \phi(r, l, T)\big) \neq 0$. $\square$

Remark 1. *Unit square domain*

$U_{(\lfloor x \rfloor, \lfloor y \rfloor)}$ has a lower left corner $(\lfloor x \rfloor, \lfloor y \rfloor)$, where $\lfloor x \rfloor = B\nu(T_k) + \nu(T_{k+1})$ and $\lfloor y \rfloor = B\nu(q) + \nu(T_{k-1})$.

Definition 8. *Left Affine Function*

Case (1) in definition 5 where $\eta(q, T_k) = (r, b, -1)$.



The left affine function is derived as follows:

$$\begin{aligned} x &\to T_{k-1} b \,.\, T_{k+1} T_{k+2} T_{k+3} \ldots \\ \tfrac{1}{B} x &= \quad T_k \,.\, T_{k+1} T_{k+2} T_{k+3} \ldots \end{aligned}$$

Subtract the numbers in base B, so $m = T_{k-1} b - T_k$.

$$\begin{aligned} y &\to \quad r T_{k-2} \,.\, T_{k-3} T_{k-4} \ldots \\ By &= q T_{k-1} T_{k-2} \,.\, T_{k-3} T_{k-4} \ldots \end{aligned}$$

Subtract the integers in base B, so $n = r T_{k-2} - q T_{k-1} T_{k-2}$. Define function $L : U_{(\lfloor x \rfloor, \lfloor y \rfloor)} \to P$ as $L(x, y) = (\frac{1}{B} x + m, By + n)$ where $m = B\nu(T_{k-1}) + \nu(b) - \nu(T_k)$ and $n = B\nu(r) - B^2 \nu(q) - B\nu(T_{k-1})$. L is called a left affine function.

Lemma 2. *Left Affine $\Longleftrightarrow$ TM Execution Step*

Let (q, k, T) be a machine configuration. Suppose $\eta(q, T_k) = (r, b, -1)$ for some state r in $Q \cup \{h\}$ and some alphabet symbol b in A and where $T_k = a$. For the next execution step, the new configuration is $(r, k-1, T^b)$ where $T^b(j) = T(j)$ for every $j \neq k$ and $T^b(k) = b$. The commutative diagram $L \circ \phi(q, k, T) = \phi \circ \eta(q, k, T)$ holds, so $L(x(q, k, T),\ y(q, k, T)) = (x(r, k-1, T^b), y(r, k-1, T^b))$.

PROOF. The x-coordinate of $L(x(q, k, T), y(q, k, T))$

$= B^{-1} x(q, k, T) + B\nu(T_{k-1}) + \nu(b) - \nu(a)$

$= B^{-1}(a T_{k+1} \,.\, T_{k+2} \ldots) + B\nu(T_{k-1}) + \nu(b) - \nu(a)$

$= a \,.\, T_{k+1}T_{k+2}T_{k+3}\cdots + B\nu(T_{k-1}) + \nu(b) - \nu(a)$
$= T_{k-1}b \,.\, T_{k+1}T_{k+2}T_{k+3}\ldots$

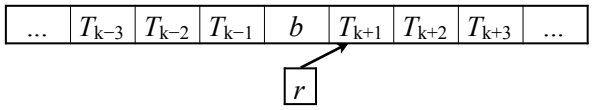
Lastly, $\eta(q, T_k) = (r, b, -1)$ implies $x(r, k-1, T^b) = T_{k-1}b \,.\, T_{k+1}T_{k+2}\ldots.$

The y-coordinate of $L(x(q,k,T), y(q,k,T))$
$= By(q,k,T) + B\nu(r) - B^2\nu(q) - B\nu(T_{k-1})$
$= B(qT_{k-1} \,.\, T_{k-2}\ldots) + B\nu(r) - B^2\nu(q) - B\nu(T_{k-1})$
$= qT_{k-1}T_{k-2} \,.\, T_{k-3}\cdots + B\nu(r) - B^2\nu(q) - B\nu(T_{k-1})$
$= rT_{k-2} \,.\, T_{k-3}T_{k-4}\ldots$
$= y(r, k-1, T^b).$

Lastly, $\eta(q, T_k) = (r, b, -1)$ implies $y(r, k-1, T^b) = rT_{k-2} \,.\, T_{k-3}T_{k-4}\ldots$ □

Definition 9. *Right Affine Function*

Case (2) in definition 5 where $\eta(q, T_k) = (r, b, +1)$.



The right affine function is derived as follows.

$$\begin{aligned} x &\rightarrow \quad T_{k+1}T_{k+2} \,.\, T_{k+3}T_{k+4}\ldots \\ Bx &= \quad T_kT_{k+1}T_{k+2} \,.\, T_{k+3}T_{k+4}\ldots \end{aligned}$$

Subtract in base B, so $m = T_{k+1}T_{k+2} - T_kT_{k+1}T_{k+2}$.

$$\begin{aligned} y &\rightarrow \quad rb \,.\, T_{k-1}T_{k-2}T_{k-3}\ldots \\ \tfrac{1}{B}y &= \quad q \,.\, T_{k-1}T_{k-2}T_{k-3}\ldots \end{aligned}$$

Subtract the numbers in base B, so $n = rb - q$. Define the right affine function $R : U_{(\lfloor x \rfloor, \lfloor y \rfloor)} \rightarrow P$ as $R(x, y) = (Bx + m, \frac{1}{B} + n)$ where $m = -B^2\nu(T_k)$ and $n = B\nu(r) + \nu(b) - \nu(q)$.

Lemma 3. *Right Affine $\iff$ TM Execution Step*

Let (q, k, T) be a machine configuration. Suppose $\eta(q, T_k) = (r, b, +1)$ for some state r in $Q \cup \{h\}$ and some alphabet symbol b in A and where $T_k = a$. For the next execution step, the new configuration is $(r, k+1, T^b)$ where $T^b(j) = T(j)$ for every $j \neq k$ and $T^b(k) = b$. The commutative diagram $R \circ \phi(q, k, T) = \phi \circ \eta(q, k, T)$ holds, so $R(x(q,k,T),\ y(q,k,T)) = (x(r, k+1, T^b), y(r, k+1, T^b))$.

PROOF. The x-coordinate of $R(x(q,k,T), y(q,k,T))$
$= Bx(q,k,T) - B^2\nu(a)$
$= B(aT_{k+1} \,.\, T_{k+2}T_{k+3}\ldots) - B^2\nu(a)$
$= aT_{k+1}T_{k+2} \,.\, T_{k+3}\cdots - B^2\nu(a)$
$= T_{k+1}T_{k+2} \,.\, T_{k+3}T_{k+4}\ldots$
$= x(r, k+1, T^b)$ because $\eta(q, T_k) = (r, b, +1)$.

The y-coordinate of $R(x(q,k,T), y(q,k,T))$
$= B^{-1}y(q,k,T) + B\nu(r) + \nu(b) - \nu(q)$
$= B^{-1}(qT_{k-1} \,.\, T_{k-2}T_{k-3}\ldots) + B\nu(r) + \nu(b) - \nu(q)$
$= q \,.\, T_{k-1}T_{k-2}T_{k-3}\cdots + B\nu(r) + \nu(b) - \nu(q)$
$= rb \,.\, T_{k-1}T_{k-2}T_{k-3}\ldots$
$= y(r, k+1, T^b)$ because $\eta(q, T_k) = (r, b, +1)$. □

Definition 10. *Halting Attractor and Halting Points*

Define *halting attractor* $\mathfrak{A}_h = \{(x, y) \in P : 0 \leq x \leq \frac{B-|A|}{B-1}B^2 \quad \texttt{and} \quad 0 \leq y \leq B-1\}$. The *halting points* in P correspond (via ϕ) to halting configurations (h, k, T). Using elementary algebra, one can verify that $\mathfrak{A}_h$ contains the halting points. Define *halting map* $\mathfrak{h} : \mathfrak{A}_h \rightarrow \mathfrak{A}_h$ such that $\mathfrak{h}(x, y) = (x, y)$. Every p in $\mathfrak{A}_h$ is a fixed point of $\mathfrak{h}$.

Definition 11. *Halting and Immortal Orbits in P*

Let $f_k : U_k \rightarrow P$ be a function for each k such that whenever $j \neq k$, then $U_j \cap U_k = \emptyset$. For any p in P, its orbit is generated as follows. The 0^{th} iterate of the orbit is p. Given the k^{th} iterate is point q, if point q does not lie in any U_k, then the orbit enters $\mathfrak{A}_h$ and halts (i.e., $\mathfrak{h}(q) = q$). Otherwise, q lies in some U_j. Inductively, the $k+1$ iterate of q is defined as $f_j(q)$. p is an *immortal point* if p has an orbit that never enters $\mathfrak{A}_h$.

Let $\{f_1, f_2, \ldots, f_I\}$ be a set of functions $f_k : X \rightarrow X$. A *function index sequence* $S : \mathbb{N} \rightarrow \{1, 2, \ldots, I\}$ indicates the order that the functions are applied. If p lies in X, then the orbit with respect to this function index sequence is $[p,\ f_{S(1)}(p),\ f_{S(2)}f_{S(1)}(p),\ \ldots,\ f_{S(m)}f_{S(m-1)}\cdots f_{S(2)}\ f_{S(1)}(p), \ldots]$.

Theorem 2. *Correspondence Theorem*

Consider machine (Q, A, η) with initial configuration $(s, 0, T)$. W.L.O.G., we assume that (Q, A, η) begins executing with its tape head location at 0. Let $p = (x(s, 0, T), y(s, 0, T))$. Per definitions 6, 8 and 9, and remark 1, let $f_k : W_k \rightarrow P$, where $1 \leq k \leq l$, be the l corresponding affine functions with unit squares W_k.

There is a one-to-one correspondence between the m^{th} point of the orbit $[p,\ \ f_{S(1)}(p),\ \ f_{S(2)} \circ f_{S(1)}(p),\ \ldots,\ \ f_{S(m)} \circ f_{S(m-1)}\cdots f_{S(2)} \circ f_{S(1)}(p), \ldots]$ and the m^{th} computational step of (Q, A, η) with initial configuration $(s, 0, T)$. Moreover, $(s, 0, T)$ is a halting (immortal) configuration if and only if p is a halting (immortal) point with respect to the halting map $\mathfrak{h}$ and affine functions $f_k : W_k \rightarrow P$ where $1 \leq k \leq l$.

PROOF. From lemmas 2, 3 and definition 6, each computational step of (Q, A, η) on current configuration (q, k, T) corresponds to applying a unique affine map f_k (definitions 8 and 9) to corresponding point $p = (x(r,k,T),\ y(r,k,T))$. By induction, the correspondence holds for all n if the initial configuration $(s, 0, T)$ is immortal; this implies that $(x(s, 0, T), y(s, 0, T))$ is an immortal point. Similarly, if $(s, 0, T)$ is a halting configuration, then machine (Q, A, η) with initial configuration $(s, 0, T)$ halts after N computational steps. For each step, the correspondence implies that the orbit of initial point $(x(s, 0, T), y(s, 0, T))$ enters $\mathfrak{A}_h$ on the N^{th} iteration. □