A Cryptographically Stable Computing Machine

Michael Stephen Fiske
Aemea Institute
mf@aemea.org

Abstract

Malware plays a critical role in breaching computer systems. The computing 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. Our primary contribution introduces a computing machine that is structurally stable to small changes made to its program instructions. Our procedures use quantum randomness to build unpredictable stable instructions. Our procedures can execute just before running a program so that the computing task can be performed with a different representation of its instructions during each run.

Our procedures are inspired by the Red Queen hypothesis in biology: organisms evolve using robustness, unpredictablity and variability to hinder infection. Another contribution expands the mathematical notion of stability to a cryptographic model with an adversary, and explains why structurally stable machines can be resistant to malware sabotage.

1. Introduction

Malware plays a critical role in breaching computer systems. Cybersecurity research has primarily focused on malware detection [1]. It seems unlikely that malware detection methods can solely provide an adequate solution to the malware problem: There does not exist a register machine algorithm that can detect all malware [2]. Furthermore, some recent malware implementations use NP problems [3] to encrypt and hide the malware [4]. Overall, detection methods are currently up against fundamental limits in theoretical computer science.

The instability of register machine computation [5] enables malware to sabotage the purpose of a computer program, by making small changes to one or more instructions in an original, uninfected program. Programming languages such as `C`, `Java`, `Lisp` and `Python` rely upon register machine [6] branching instructions. One sabotaged branching instruction enables malware to start executing. Even if there is a routine to verify that the program is executing properly, this verification routine may never execute. *Sequential execution of unstable register machine instructions cripples the program from protecting itself* [7][1].

Prior mathematical research has not attempted to design malware resistant computation based on structural stability. For over 80 years, dynamical systems has extensively studied structural stability [8, 9] on phase spaces [10], containing an uncountable number of states [11][2]. During execution of a register machine, the machine's state at any moment lies in a discrete space, containing a countable number of states.

Based on dynamical systems and information theory, our primary contribution develops mathematical and computational tools to build a structurally stable sequential machine that is resistant to small changes to the program instructions. Our approach is inspired by the Red Queen hypothesis [12] in evolutionary biology: organisms evolve using robustness, unpredictability and variability to hinder infection from parasites. Another contribution expands the notion of stability to a cryptographic model with an adversary, and explains why this structurally stable machine is resistant to malware sabotage.

2. Motivating Stable Computation

Register machines execute one instruction at a time. Even if there is a procedure to assure that the register machine program is executing correctly, *this friendly procedure may never execute due to just one rogue branch instruction.* Typical programming languages (e.g., `C`, `Fortran`, `Java`, `Lisp` and `Python`) are Turing complete and depend upon branching

[1] Non-register machines, such as the Active Element Machine [7], can execute multiple machine instructions simultaneously.

[2] A space X is uncountable if X contains an infinite number of states and there does not exist a 1-to-1 correspondence between X and the natural numbers $\mathbb{N}$. A space X is countable if there exists a 1-to-1 correspondence between X and $\mathbb{N}$.

URI: https://hdl.handle.net/10125/80251
978-0-9981331-5-7

instructions. While conditional branching instructions are not required for universal computation, Rojas's methods [13] 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.[3]

These observations suggest that a computer program's purpose can be subverted because the register machine behavior is not always invariant when small changes are made to one or more instructions.

Overall, we seek stable computation based on the following design principle: if a small or moderate change is made to a register machine program, then the program's purpose is stable; if a large change is made, the program can no longer execute. Our principle is partly based on the observation that it is generally far more difficult to detect if a small change has altered the purpose of a program. With a small change, the tampered register machine program still can execute, but does not perform the task that the original program was designed to accomplish. For this reason, our goal is to create stable computation that is also incomprehensible to malware authors so that it is far more challenging for malware to subvert the program without completely destroying its functionality.

3. An Unstable C Program

We demonstrate *unstable computation* with C source code [14] that adds 3 integers. This C code shows how a 1-bit change to the address of only one instruction can substantially alter the program's behavior.

```
#include <stdio.h>

#define NUM_BITS  16
int pow2[NUM_BITS] = {0x8000, 0x4000, 0x2000, 0x1000,
                      0x800, 0x400, 0x200, 0x100,
                      0x80, 0x40, 0x20, 0x10,
                      0x8, 0x4, 0x2, 0x1};


int addition(int  a,  int b)
{
   return (a + b);
}

int multiply(int  a,  int b)
{
   return (a * b);
}

int exec_op(int* num, int n, int (*op) (int, int))
{
   int  i, v = num[0];

   for(i = 1; i < n; i++)
   {
      v = op(v, num[i]);
   }

   return v;
}
```

[3] See figure A.14 in [5].

```
void print_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 / pow2[k])  printf("1 ");
      else              printf("0 ");

      v %= pow2[k];
   }
   printf("\n");
}

int bop(int* m, int n, char* f, int (*op) (int, int))
{
   int  v = exec_op(m, n, op);

   printf("\nresult = %d. address of ", v);
   printf("%s  = \n", f);
   print_binary((unsigned int) op);

   return 0;
}

int main(int argc, char* argv[])
{
   int num[3] = {2, 3, 5};

   print_numbers(num, 3);
   printf("\n");
   bop(num, 3, "addition", addition);
   bop(num, 3, "multiply", multiply);

   return 0;
}
```

```
aemea@Michaels-MacBook-Air C_program %  ./ADD

2 3 5

result = 10. address of addition  =
1 0 1 0 1 1 0 0 0 1  0  1 0 0 0 0

result = 30. address of multiply  =

1 0 1 0 1 1 0 0 0 1  1  1 0 0 0 0
```

Figure 1. Sum Changed to a Product

Figure 1 shows an execution of the compiled C program: ADD. A sum $2 + 3 + 5$ is converted to a product $2 * 3 * 5$, by flipping only one bit of the address of instruction addition. This C program exhibits unstable computation because a small change (flipping one bit) in the C program causes a substantial change to the outcome: namely, a sum equal to 10 is changed to a product equal to 30.

4. Cryptographic Model Assumptions

In our model, *Alice*'s goal is to hinder *Eve* (malware author) from sabotaging Alice's computation. Additional assumptions about what information Eve has access to are more appropriate to discuss after a comprehensive description of our structurally stable machine is provided. Some of these assumptions defer prospective hardware implementations to a subsequent paper. *Bob* is not part of our model. We do not address commmunication between Alice and Bob, as is typical with a public key exchange.

5. Structural Stability

Structural stability is a mathematical tool that can be applied to computer programs because a register machine program can be modeled as a discrete, autonomous dynamical system [15]. When a perturbed instruction is close enough to an original instruction that is stable, then the computational behavior of the program will not change. For this reason, structural stability can be used to design a solution that hinders malware sabotage. We briefly review topological spaces, metric spaces and structural stability.

5.1. Topological Spaces & Metric Spaces

A *topology* [11] 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 $\mathcal{T}$. The standard topology on $\mathbb{R}$ (real numbers) is generated from arbitrary unions of open intervals and finite intersections of open intervals, where an open interval is $(a, b) = \{x \in \mathbb{R} : a < x < b\}$.

For topological spaces X (domain) and Y (range), a function $f : X \to Y$ is *continuous* if for any open subset U of Y, the inverse image $f^{-1}(U) = \{x \in X : f(x)$ lies in $U\}$ is open in X's topology. A function $h : X \to Y$ is a *homeomorphism* if h is continuous, h is bijective and h's inverse $h^{-1} : Y \to X$ is continuous.

A *metric space* is a set X and a function (metric) $d : X \times X \to \mathbb{R}$ such that all three conditions hold: (1) $d(a, b) \geq 0$ for all $a, b \in X$ where $d(a, b) = 0$ if and only if $a = b$. (2) $d(a, b) = d(b, a)$ for all $a, b \in X$. (3) $d(a, b) \leq d(a, c) + d(c, b)$ for all $a, b, c \in X$.

5.2. Topological Conjugacy & C^0 Stability

A *discrete, dynamical system* is a function $f : X \to X$, where X is a topological space. Two dynamical systems $f : X \to X$ and $g : Y \to Y$ are *topologically conjugate* if f and g are continuous and there exists a homeomorphism $h : X \to Y$ such that $h \circ f = g \circ h$.

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, f is structurally stable if for all dynamical systems g that are close to f, then f is topologically conjugate to g.

After a register machine has halted, its halted machine configuration represents what the machine has computed. Topological conjugacy is useful because each halted machine configuration corresponds to a fixed point (halting point) of a dynamical system that faithfully models the register machine. 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 M_1 and M_2 induces a 1-to-1 correspondence between the halting configurations of M_1 and M_2.

6. A Structurally Stable Machine

We have two design goals, motivated by section 2:

A. Build instructions that are invariant to small changes in their representation.

B. Hinder the adversary from figuring out how to manipulate these machine instructions.

We start with a Turing complete, virtual machine, as a starting point for building computation that satisfies our two design goals. We describe procedures[4] that transform these virtual machine instructions: these transformation procedures represent and execute the functionality of the virtual machine instructions so that the computation is stable under small changes to the transformed instructions. Furthermore, these transformed instructions also satisfy design goal B.

6.1. A Virtual Register Machine

Below is a brief description of the instructions for our virtual machine. Our transformation procedures,

[4]Comprehensive hardware implementation(s) of these procedures are beyond the scope of this paper.

applied to these virtual machine instructions, are described in subsections 6.2, 6.4, 6.5 and 6.6.

`SET A 14` stores 14 in register `A`.

`ADD B C` adds the contents of registers `B` and `C` and stores the sum in register `B`.

`SUB C 13` subtracts 13 from the contents of register `C` and stores the difference in register `C`.

`MUL D C` multiplies the contents of register `D` and register `C` and stores the product in register `D`.

`DIV A 7` divides the contents of register `A` by 7 and stores the quotient in register `A`. If `A` contains 26, then after `DIV A 7` is executed register `A` contains 3.

`MOD A 7` divides the contents of register `A` by 7 and stores the remainder in `A`. If register `A` contains 26, then after `MOD A 7` is executed register `A` contains 5.

`JMP L_AB` updates the program counter to execute the instruction tagged by label `L_AB`. If `L_AB` contains 37, then instruction 37 will be executed next. `JMP` acts as an unconditional branch instruction.

`IF A B` executes the next instruction if the contents of register `A` equals the contents of register `B`. Otherwise, the next instruction is skipped.

`IFN C 5` executes the next instruction if the contents of register `C` are not equal to 5. If register `C` contains 5, the next instruction is skipped.

`IFGT C 12` executes the next instruction if the contents of register `C` are greater than 12. If register `C` contains a number less than 13, the next instruction is skipped.

The *opcode* for *instruction name* `SET` represents `SET` in terms of bits. The other instruction names {`ADD`,`SUB`,`MUL`,`DIV`,`MOD`,`JMP`,`IF`,`IFN`,`IFGT`, `STORE`, `GET`} each have their own unique opcode in terms of bits. An opcode is mathematically defined as a function $\mathfrak{O}$: {`SET`,`ADD`,`SUB`,`MUL`,`DIV`,`MOD`, `JMP`,`IF`,`IFN`,`IFGT`,`STORE`,`GET`} $\rightarrow \{0,1\}^n$ such that $\mathfrak{O}$ is 1-to-1. 1-to-1 means $\mathfrak{O}$ maps two different instruction names to two distinct n-bit strings.

A valid instruction starts with an instruction name, followed by one or two *operands*. In instruction `IFGT C 12`, register `C` is the first operand and the number `12` is the second operand. The `JMP` instruction is the only instruction with one operand.

In subsection 6.6, example 1 implements a greatest common divisor algorithm with this virtual machine. In subsection 6.2, we develop computational tools that transform these instructions so that the program instructions are hidden, stable and unpredictable.

6.2. Randomizing Instruction Opcodes

We describe a procedure that randomizes opcodes such that each opcode is a minimal Hamming distance apart. Our random opcode procedure is a computational tool for helping us achieve our two design goals.

First, we review some definitions from information theory. $\{0,1\}^n$ is the collection of all n-bit strings, where each binary string a in $\{0,1\}^n$ can represent an opcode of a virtual machine instruction. Sometimes b in $\{0,1\}^p$ can represent an operand of a virtual machine instruction, and in some cases $n \neq p$.

Let $a = a_1, \ldots a_n$ and $b = b_1, \ldots b_n$ be binary strings of length n. For each n, the *Hamming metric* [16] is defined as $d(a,b) = \sum_{i=1}^{n} |a_i - b_i|$. It is easy to verify that $(\{0,1\}^n, d)$ is a metric space per section 5.1.

Note $d(0010, 0111) = 2$. Consider string $c = c_1 \ldots c_n$ in $\{0,1\}^n$. A *Hamming ball* $H(c,m) = \{a \in \{0,1\}^n : d(c,a) \leq m\}$ has center c and radius m.

Let q be a quantum random bit generator[5] [17]. Based on a quantum measurement, q returns a random 0 or 1. In procedures 1, 2, and 3, q helps construct m distinct random opcodes each of length n that are pairwise a minimum Hamming distance of $2l + 1$ bits apart. These procedures build random opcodes that are stable, when there are at most l bits of sabotage on a single opcode. A random opcode I_j is the center of a Hamming ball with radius l. Geometrically, all opcodes in $H(I_j, l)$ can be repaired to the correct opcode I_j.

Procedure 1 builds an n-bit random opcode $I_{j,1} I_{j,2} \ldots I_{j,n}$ used by instruction I_j, after quantum random bit generator q measures n random bits.

Procedure 1. *Random Opcode*

```
Input: n
set k := 1
while  k ≤ n
{
   set I_{j,k} to a random bit measured by q
   increment k by 1
}
Output: I_{j,1} I_{j,2} ... I_{j,n}
```

The jth random opcode, called I_j, is n bits long. Procedure 2 begins with m random opcodes I_1, I_2, ..., I_m as input. Procedure 2 finds two distinct opcodes in

[5] We prefer a QRNG over a CSRNG based on the following principle: no sentient being can capture or steal information that does not yet exist. According to quantum theory, a quantum random bit does not exist until a measurement occurs. A CSRNG's effectiveness depends upon on an algorithm: any Turing machine, implementing the algorithm, violates our principle, and relies on a seed. A CSRNG provides no unpredictability if the seed is known. A CSRNG also begs the question: how is an unpredictable seed generated?

$\{I_1, I_2, \ldots, I_m\}$ that are a smallest Hamming distance h_μ apart.

Procedure 2. *Minimal Hamming Pair*

```
Input: m, n, and opcodes {I_1, I_2, ..., I_m}
set h_μ := n + 1
set i := 1
while i < m
{
   set j := i + 1
   while j ≤ m
   {
      set h := d(I_i, I_j) in {0,1}^n
      if (h < h_μ)
      {
         set h_μ := h
         set i_μ := i
         set j_μ := j
      }
      increment j by 1
   }
   increment i by 1
}
Output: I_{i_μ}, I_{j_μ}, h_μ
```

Procedure 3 uses procedures 1 and 2 to build m random opcodes (n-bit codes) that are pairwise at least a Hamming distance of $2l + 1$ apart.

Procedure 3. *Minimal Hamming Distance*

```
Input: l, m, n

call procedure 1 m times with input n :
output I_1, ..., I_m

call procedure 2 with input I_1, ..., I_m :
output I_{i_μ}, I_{j_μ}, h_μ

set r := 0
while h_μ < (2l + 1)
{
   set b := 2l + 1 − h_μ
   do b times
   {
      use q to randomly choose positive
      integer k in the set {1, 2, ..., n}

      flip bit k in I_{i_μ}
   }
   execute procedure 2 with input I_1, ..., I_m
   increment r by 1
}
Output: I_1 ... I_m with d(I_j, I_k) ≥ 2l+1 when j ≠ k
```

Flip bit k means: if the kth bit is 0, then set the kth bit to 1; and if the kth bit is 1, then set the kth bit to 0. Variable r counts the number of repairs on a random opcode until any two distinct opcodes are at least a distance of $2l + 1$-bits apart. $2l + 1$ should be about 1 to 2.5 standard deviations less than $\frac{n}{2}$ so that outer loop `while` $h_\mu < (2l + 1)$ promptly exits.[6]

If l is too large (e.g., $(2l + 1) > n$), then the outer loop never exits and $r \to \infty$. To avoid long computing times, a variation of procedure 3 inserts, before the outer loop, `set` $l = \lfloor \frac{1}{4}(n - c\sqrt{n}) - \frac{1}{2} \rfloor$, where $1 \le c \le \frac{5}{2}$.

In our cryptographic model, Alice's m random opcodes $I_1, I_2, \ldots I_m$ "act as her private keys." Hence, her random opcodes should be generated and stored in Protected Machine Hardware (blue region in Figure 2) so that procedure 3 can help assure anonymity and valid execution of her instructions. It is also good practice for Alice to keep l private.

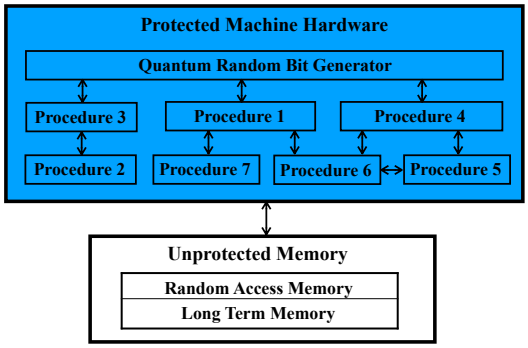

Figure 2. Eve cannot access the blue region.

6.3. Instruction & Program Stability

We formally define instruction and program stability.

Definition 1. *A set of opcodes (or operands)* $\{I_1, \ldots, I_m\}$ *is s-bit stable if* $\min\{d(I_j, I_k) : j \neq k\} \geq s$. *In other words, if* I_j *and* I_k *are the closest opcodes (operands) in* $\{I_1, \ldots, I_m\}$, *then* $d(I_j, I_k) \geq s$.

Definition 2. *An* n*-bit instruction* I *is* s*-bit stable if its opcode and operands are both* s*-bit stable.*

Remark 1. The random opcodes generated by a successful exit of procedure 3 are $2l + 1$-bit stable.

Procedure 2 returns h_μ. h_μ is the minimum distance between any two distinct opcodes in $\{I_1, \ldots, I_m\}$. A successful exit of procedure 3 means the loop `while` $h_\mu < (2l + 1)$ exited; hence, $h_\mu \geq 2l + 1$.

[6] $\frac{n}{2}$ is the expected Hamming distance between two random n-bit codes, where each bit occurs with probability $\frac{1}{2}$. The standard deviation of uniformly random n-bit codes is $\frac{\sqrt{n}}{2}$.

Using definitions 1, 2 and remark 1, we can explain the stability of programs, transformed by procedures 1-3, in terms of the theory of section 5. First, we need to define a metric on a space of programs, where each program is a finite sequence of transformed (procedures 1-3) instructions (before hiding). Consider a program of transformed instructions $\mathcal{P}_1 = (I_1, I_2, \ldots, I_m)$ and another program of transformed instructions $\mathcal{P}_2 = (J_1, J_2, \ldots, J_m)$. If two programs have the same length m, define the distance between them as $\mathcal{D}(\mathcal{P}_1, \mathcal{P}_2) = \max\{d(I_k, J_k) : 1 \leq k \leq m\}$. If programs $\mathcal{P}$ and $\mathcal{Q}$ have different lengths, then define $\mathcal{D}(\mathcal{P}, \mathcal{Q}) = n$, where all program instructions in $\mathcal{P}$ and $\mathcal{Q}$ lie in $\{0,1\}^n$.

Assume each instruction I_k, for $1 \leq k \leq m$, in $\mathcal{P}_1 = (I_1, I_2, \ldots, I_m)$ is $2l+1$-bit stable. Consider $\mathcal{P}_1$ as a dynamical system. Program $\mathcal{P}_1$ is structurally stable because whenever $\mathcal{D}(\mathcal{P}_1, \mathcal{P}_2) \leq l$ each instruction J_k in $\mathcal{P}_2$ that corresponds to I_k is resolved to the same opcode and same operand(s). Thus, the dynamical (computing) behavior of $\mathcal{P}_1$ is the same as program $\mathcal{P}_2$.

6.4. Hiding Operands and Opcodes in Noise

In 6.2, we provided procedures 1, 2, and 3 for building opcodes stable to small changes. When these procedures are also applied to operands, the representation of the operands can become stable to small changes. In our model, Eve is a sentient adversary, so structural stability from classical mathematics alone does not provide enough mathematical firepower to build malware resistant computation. Hence, the purpose of design goal B is to build a representation of each instruction that is computationally intractable for Eve to understand its meaning.

Procedure 4 builds a permutation based on [18, 19].

Procedure 4. *Random Permutation ρ*

Input: n

set $\rho(1) := 1$ set $\rho(2) := 2$ ... set $\rho(n) := n$

set $k := n$

while $k \geq 2$
{
  use q to randomly choose positive integer r in the set $\{1, \ldots, k\}$

  set $t := \rho(r)$

  set $\rho(r) := \rho(k)$

  set $\rho(k) := t$

  decrement k by 1
}

Output: Permutation ρ on $\{1, \ldots, n\}$

Procedure 5 uses procedure 4 to build a random substitution box.

Procedure 5. *Random Substitution Box σ*

Input: η

call procedure 4 with input $n = 2^\eta$ to create random permutation σ

Output: Substitution Box σ
σ maps an η-bit input to an η-bit output

The rest of this subsection describes how to hide the meaning of the opcodes and operands from Eve. Procedures 4 and 5 have different purposes even though they both produce a random permutation. Procedure 5 constructs a σ that has size 2^η. Typically, $\eta = 8$ because 8 bits is a byte, and $n = 16$ is too large.[7] If the operands in the base virtual machine have size equal to 64 bits before hiding in 64 bits of noise, then procedure 5 is called sixteen times to generate $\sigma_1\ \sigma_2\ \ldots\ \sigma_{16}$ for the first operand. For the second operand, procedure 5 is called sixteen more times to generate $\sigma_{17}\ \ldots\ \sigma_{32}$. In general, σ_i and σ_j are statistically independent when $i \neq j$, as result of using procedure 5, based on [19].

Procedure 4 builds ρ to locate the 64 bits of the signal $b_1 b_2\ \ldots\ b_{64}$ inside the random noise. ρ lies in the symmetric group on $\{1, 2, \ldots, n\}$, and determines where each bit of the signal (i.e., opcode or operand) is located: the ith bit b_i is stored at bit location $\rho(i)$, where $1 \leq i \leq 64$. When there are 64 bits of signal from the operand or opcode and 64 bits of quantum random noise, then the size of ρ is 128, i.e., $n = 128$.

Random noise is measured and stored in the remaining 64 bit locations. The result is 128 bits of noise and signal, named $s_1 \ldots s_{128}$. Subsequently, the 16 randomly generated sboxes σ_i with $1 \leq i \leq 16$ are applied to $s_1 s_2 \ldots s_{128}$ as follows: $\sigma_1(s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8)$, $\sigma_2(s_9 \ldots s_{16})$, $\ldots$ $\sigma_{16}(s_{121} s_{122} s_{123} s_{124} s_{125} s_{126} s_{127} s_{128})$, which is named $c_1 c_2 \ldots c_{128}$. Next, a distinct random permutation τ is generated on $\{1, 2, \ldots, 128\}$. τ is applied to $c_1 c_2 \ldots c_{128}$, resulting in $c_{\tau(1)}\ c_{\tau(2)}\ \ldots\ c_{\tau(128)}$. Then sixteen distinct sboxes $\alpha_1\ \ldots\ \alpha_{16}$ are randomly generated and applied to $c_{\tau(1)} c_{\tau(2)} \ldots c_{\tau(128)}$ as follows: $\alpha_1(c_{\tau(1)} c_{\tau(2)} \ldots c_{\tau(8)})\ \ldots\ \alpha_{16}(c_{\tau(121)} \ldots c_{\tau(128)})$, resulting in $o_1 \ldots o_{128}$.

In subsection 7.2, a birthday paradox statistical test is described in order to address potential attacks that involve stable instructions of size 128 bits. The birthday attack test performs 2^l compilations of the same unmasked instruction. In 7.2, we also describe a statistical test to address multiple transformations via procedure 6 of the same instruction at different locations in a single SVM compilation of the program.

[7] If $\eta = 16$, then σ has size $2^{16} = 65536$. If $\eta = 16$, then σ is considerably more expensive than $\eta = 8$. Our goal is to store σ in protected machine hardware (Figure 2) that is not accessible to Eve.

For each time the SVM compilation tool is executed, all ρ, σ_i, τ, α_i and randomized opcodes are statistically independent from each other and from all previous compilations. Also, within one SVM compilation, the random noise ($n-k$ bits per opcode) generated for two identical opcodes at different locations in the program is statistically independent.

From our prior description, procedure 6 formally specifies hiding a k-bit opcode (or operand) $b_1 \dots b_k$ in $n-k$ bits of random noise. ρ and τ are distinct random permutations on $\{1, 2, \dots, n\}$. ρ determines the bit locations of $b_1 \dots b_k$ hidden inside of noise. $\sigma_1 \dots \sigma_p$ and $\alpha_1 \dots \alpha_p$ are random sboxes.

Procedure 6. *Hide in Noise*

```
Input: k, n and k-bit string b_1 b_2 ... b_k
call procedure 4 with input n to build
random permutation ρ on {1, 2, ..., n}

call procedure 1 with input n and store
noise in every bit location l_1 l_2 ... l_n

set j := 1
while j ≤ k
{
   set bit location l_ρ(j) := b_j
   increment j by 1
}
call procedure 5 p times on input η and
generate substitution boxes σ_1 ... σ_p

apply substitution boxes σ_1 ... σ_p to input
l_1 l_2 ... l_n and compute c_1 c_2 ... c_n

call procedure 4 with input n to build
random permutation τ on {1, 2, ..., n}

permute c_1 ... c_n to c_τ(1) ... c_τ(n)

call procedure 5 p times on input η and
generate substitution boxes α_1 ... α_p

apply α_1 ... α_p to input c_τ(1) ... c_τ(n)

Output: o_1 o_2 ... o_n
```

6.5. Hiding Instruction Order

Procedure 7 hides the hidden stable instructions, constructed by procedure 6, inside a block of size b containing dummy instructions. b is the sum of the number of stable instructions and the number of dummy instructions in the block. m is the maximum number of stable instructions hidden. γ is a random permutation on $\{1, 2, \dots, b\}$. S_1, S_2, ..., S_m are a sequence of stable instructions that are part of the program.

Procedure 7. *Hide Stable Instructions in Block*

```
Input: b, m, γ, and S_1, S_2, ... and S_m
use q to randomly choose a positive
integer c in the set {1, 2, ..., ⌊b/2⌋}
set i := 1   set j := 1   set k := 1
while i ≤ b
{
  use q to randomly choose a positive
  integer r in the set {1, 2, ..., b}
  if ( (r ≤ c) and (j < m) )
  {
     hide instruction S_j at location γ(i)
     increment j by 1
  }
  else
  {
     randomly build dummy instruction D_k
     hide instruction D_k at location γ(i)
     increment k by 1
  }
  increment i by 1
}
```

Table 1 shows a representation of the stable hidden instructions permuted in the block with dummy instructions after procedure 7 is completed.

Table 1. Hiding Instruction Order

Block Index	Instr. Name	Instr. Type	Test $r \le c$	Gamma Index
1	S_{j_1}	Valid	True	$\gamma(j_1)$
2	D_{r_1}	Dummy	False	$\gamma(r_1)$
...				
$\gamma^{-1}(1)$	S_1	Valid	True	$\gamma(1)$
...				
b	$D_{r_{k-1}}$	Dummy	False	$\gamma(r_{k-1})$

In table 1, $j_i = \gamma^{-1}(i)$ and $r_i = \gamma^{-1}(i)$.

Because c is randomly selected in $\{1, \dots \lfloor \frac{b}{2} \rfloor\}$, and m is determined during compilation and m can be randomly selected for each block, Eve does not know how many valid instructions are in a block. Eve does not know the probability distribution of valid versus dummy instructions in a block; and Eve does not know the block size. Eve does not know γ, and she does not know how to distinguish a dummy opcode from a valid opcode, since the dummy opcodes are also selected using procedure 1.

In our cryptographic model, Alice's Protected Machine Hardware (Figure 2) should execute the operations in procedures 6 and 7 because the sboxes $\sigma_1 \dots \sigma_p$ and $\alpha_1 \dots \alpha_p$ and permutations ρ, τ, γ "act as Alice's private keys."

6.6. Executing Stable, Hidden Instructions

After the inverse of procedure 7 is performed on a block of the program, executing a stable, hidden instruction consists of 3 steps.

1. Unmask and find the nearest valid opcode.
2. Extract the operands from the noise.
3. Execute a valid instruction.

In our cryptographic model, we assume there is at least one hardware implementation where all three steps are executed in Protected Machine Hardware (blue region in Figure 2). Our model assumes Eve does not have access to the internal physical operations in the blue region during the execution of these three steps.

With our model assumption, we proceed to examine steps 1, 2, and 3 in more detail. We developed a software tool, called SVM, in `ANSI C` [14] that performs these 3 steps. Our SVM tool uses the quantum random bit generator in [20]. In general, our SVM tool can execute any hidden program that operates according to the virtual register machine instructions, described in section 6.1. After a brief summary of steps 1, 2, and 3, we demonstrate an example of our SVM tool, building hidden, stable virtual register machine instructions.

We assume that our cryptographically stable instructions are stored in Unprotected Memory (Figure 2). After a block of hidden instructions are retrieved from Unprotected Memory, they are ordered, unmasked and executed in the Protected Machine Hardware. In step 1, the first argument of the instruction is a noisy opcode. Our SVM tool finds the nearest valid opcode to the noisy opcode, by computing the Hamming distance between the noisy opcode and valid opcodes. If the nearest opcode is a dummy opcode, the instruction is ignored; otherwise, in step 2, the operands are extracted from the noise by executing a procedure that performs the inverse of procedure 6. In step 3, a valid opcode executes with unmasked operands as input.

Example 1. Unmasked GCD Program

Symbols, following a semicolon on the same line, are comments.

```
SET   A   6  ; Instruction 0
SET   B  10  ; Instruction 1
IF    A   0  ; 2. If (A == 0) execute instruction 3.
JMP  12      ; 3. Branch to instruction 12.
IF    B   0  ; 4. If (B == 0) execute instruction 5.
JMP  13      ; 5. Branch to instruction 13.
IFGT  A   B  ; 6. If (A > B) execute instruction 7.
SUB   A   B  ; 7. Store A-B in register A.
IFGT  B   A  ; 8. If (B > A) execute instruction 9.
SUB   B   A  ; 9. Store B-A in register B.
IFN   A   B  ; 10. If (A != B) execute instruction 11.
JMP   2      ; 11. Branch to instruction 2.
SET   A   B  ; Instruction 12
SET   B   A  ; Instruction 13
```

GCD Instructions Executed:

```
SET   A  6
SET   B  10
IF    A  0
IF    B  0
IFGT  A  B
IFGT  B  A
SUB   B  A
IFN   A  B
JMP   2
IF    A  0
IF    B  0
IFGT  A  B
SUB   A  B
IFGT  B  A
SUB   B  A
IFN   A  B
SET   A  B
SET   B  A
```

After the last instruction `SET B A` executes, both registers `A` and `B` are storing 2. Instruction numbers in an SVM program always start at 0, so `JMP 2` causes the SVM to execute `IF A 0`. In the next section, we analyze what our stable instructions look like to Eve in Unprotected Memory after multiple SVM compilations.

7. Complexity, Statistics, & Performance

We estimate the complexity of our hiding procedure, and estimate memory use and computing time.

7.1. Complexity Estimate

We use the same parameter values as in 6.4. For larger registers, the complexity scales favorably because $f(n) = n!$ grows much faster than $e(n) = 2^n$. Calculating $\frac{f(n)}{e(n)}$ with sizes 128 and 256 bits: $\frac{f(128)}{e(128)} \in [10^{177}, 10^{178}]$ and $\frac{f(256)}{e(256)} \in [10^{430}, 10^{431}]$. $f(n)$ is compared to binary exponential growth $e(n)$ because procedures 4, 5 and 6 use permutations; standard cryptographic methods rely on the $P \neq NP$ assumption that complexity grows exponentially. In procedure 6, each 64-bit opcode ($k = 64$) is hidden in 64 bits of noise ($n = 128$). There are $128 * 127 \ldots 66 * 65 > 10^{126}$ locations for hiding a 64-bit operand in 64 bits of noise.

7.2. Statistical Testing for Two Attacks

The first test searches for a birthday paradox attack with distinct SVM compilations of a fixed 64-bit random opcode in the unmasked instruction `SET A 6`.

We used $n = 64$ for procedure 1 to generate the random opcode for `SET`, and inputs $l = 12$, $m = 12$, and $n = 64$ for procedure 3. We used input $n = 128$ for procedure 4; and input $\eta = 8$ for procedure 5; and inputs $k = 64$ and $n = 128$ for procedure 6. We searched for collisions, where 75% of the bits match: that is,

where the Hamming distance was greater than 96 for two opcode portions of the instruction `SET A 6`.

The statistics for 44850 Birthday paradox collision search comparisons are shown below, where each tally is the hamming distance between two 128-bit hidden opcodes, generated from instruction `SET A 6`. Let h_i be the number of pairs of distinct stable instructions whose 128 bits of opcode are a Hamming distance of i bits apart. Below is the data for a typical run with 300 stable instructions generated from `SET A 6`:

TOTAL number of h_i computed = 44850.

$h_{40} = 1$. $h_{41} = 1$. $h_{42} = 0$. $h_{43} = 2$. $h_{44} = 4$. $h_{45} = 6$. $h_{46} = 14$. $h_{47} = 33$. $h_{48} = 64$. $h_{49} = 87$. $h_{50} = 150$. $h_{51} = 237$. $h_{52} = 328$. $h_{53} = 460$. $h_{54} = 667$. $h_{55} = 902$. $h_{56} = 1114$. $h_{57} = 1451$. $h_{58} = 1809$. $h_{59} = 2142$. $h_{60} = 2516$. $h_{61} = 2795$. $h_{62} = 2927$. $h_{63} = 3068$. $h_{64} = 3157$. $h_{65} = 3184$. $h_{66} = 2929$. $h_{67} = 2729$. $h_{68} = 2492$. $h_{69} = 2100$. $h_{70} = 1809$. $h_{71} = 1522$. $h_{72} = 1115$. $h_{73} = 929$. $h_{74} = 676$. $h_{75} = 480$. $h_{76} = 361$. $h_{77} = 219$. $h_{78} = 145$. $h_{79} = 93$. $h_{80} = 57$. $h_{81} = 38$. $h_{82} = 18$. $h_{83} = 12$. $h_{84} = 1$. $h_{85} = 6$. When $i > 85$, $h_i = 0$.

Empirical mean = 64.02. μ = 64. Expected standard deviation $\sigma = 5.66$.
30039 h_i are within σ of μ. Expected h_i within σ = 30615.
42973 h_i are within 2σ of μ. Expected h_i within 2σ = 42805.
44714 h_i are within 3σ of μ. Expected h_i within 3σ = 44733.

No Hamming distances were close to 96 (75% of 128).

In test 2, we searched for an attack by looking at multiple compilations of the same instruction at different locations in a single SVM compilation of the program. To simplify this search and make an attack easier to find, we built a program with 150 identical instructions: `SET A 6`. After this single compilation, we computed pairwise Hamming distances. Statistics for a typical run of test 2 are shown below:

TOTAL number of h_i computed = 11175.

$h_{43} = 4$. $h_{44} = 2$. $h_{45} = 3$. $h_{46} = 4$. $h_{47} = 7$. $h_{48} = 15$. $h_{49} = 30$. $h_{50} = 43$. $h_{51} = 43$. $h_{52} = 75$. $h_{53} = 122$. $h_{54} = 167$. $h_{55} = 237$. $h_{56} = 309$. $h_{57} = 389$. $h_{58} = 467$. $h_{59} = 520$. $h_{60} = 648$. $h_{61} = 667$. $h_{62} = 701$. $h_{63} = 753$. $h_{64} = 867$. $h_{65} = 782$. $h_{66} = 697$. $h_{67} = 670$. $h_{68} = 655$. $h_{69} = 535$. $h_{70} = 437$. $h_{71} = 345$. $h_{72} = 259$. $h_{73} = 223$. $h_{74} = 168$. $h_{75} = 117$. $h_{76} = 82$. $h_{77} = 46$. $h_{78} = 37$. $h_{79} = 23$. $h_{80} = 10$. $h_{81} = 7$. $h_{82} = 6$. $h_{83} = 1$. $h_{84} = 1$. $h_{85} = 1$. When $i > 85$, $h_i = 0$.

Empirical mean = 63.90. μ = 64. Expected standard deviation $\sigma = 5.66$
7495 h_i are within σ of μ. Expected h_i within σ = 7628.
10735 h_i are within 2σ of μ. Expected h_i within 2σ = 10665.
11139 h_i are within 3σ of μ. Expected h_i within 3σ = 11146.

No Hamming distances were close to 96; the statistics follow a binomial distribution with $p = \frac{1}{2}$. In general, we do not expect this attack to be effective for Eve because Alice should not voluntarily compile her plaintext code with static "keys" on plaintext code that repeats the same instruction multiple times on purpose. Furthermore, Eve's potential collisions occur only on "keys" that Eve artificially constructs with the SVM tool. In a proper use setting, Alice uses a different set of "keys" and randomized opcodes on each separate compilation on a particular machine, and procedure 7 further reduces the efficacy of this type of attack.

7.3. Hardware Performance Estimates

In section 6.4, 64-bit operands (or opcodes) are hidden in 64 bits of noise ($k = 64$, $n = 128$). If we assume that we are not utilizing 56 out of the 64 bits that represent the opcode, then our memory use in unprotected memory increases linearly at most by $4x$.

We use Shi and Lee [21] as a reference to estimate computing speeds of our bit permutations executed in Protected Machine Hardware (procedures 4 and 6). Shi and Lee rigorously analyze implementing arbitrary bit permutation operations in hardware. Their complexity estimates are expressed in terms of logical effort [22]. Logical effort can be used to estimate the number of stages required to implement the critical path of a given logic function with CMOS, and determine the maximum possible speed of the circuit.

In [21], they found that the Butterfly network is the fastest architecture for implementing bit permutations. For a 6-stage Butterfly network they found a latency of 12.0 FO4 [23]. Microprocessors typically have a cycle of time of 20-30 FO4, so the Butterfly network should be able to complete all bit permutation operations in 1 or 2 instruction cycles. The generation of the quantum random bits can be performed offline to support procedures 1-7. (The inverses of procedures 6 and 7 do not require a quantum random bit generator.)

8. Related Work

In [24], they present a general approach to addressing code-injection attacks in scripting and interpreted languages (e.g., web-based SQL injection), by randomizing the instructions. In [24], they do not address malware attacks at the machine instruction (physical hardware) level; there is no notion nor use of stability to build instruction opcodes and operands that are resistant to small changes.

Fully homomorphic encryption (FHE) [25] is a clever method for protecting Alice's computations on data in the cloud. However, FHE does not use stability, and does not protect the computer instructions that store the private FHE key. FHE defers the malware problem to the user's local computer; FHE does not address how to hinder malware on the local computer. Some FHE keys are 1 Gigabyte and the plaintext-to-ciphertext expansion is 10,000 to 1 for just 100 bits of security [26, 27]. FHE's huge memory requirements are not currently economically feasible for protecting instructions in hardware. From section 7, we see that procedures 1-7 surpass FHE by many orders of magnitude when one compares the amount of complexity obtained for a given amount of memory and computing speed.

Secure multi-party computation (MPC) [28, 29] enables a group to jointly perform a computation without disclosing any participant's private inputs. MPC does not address when Alice does not trust her own machine, or the machine instructions being executed on her machine. With further research, it is conceivable that the stability and / or hiding methods described herein could be integrated into hardware with an augmentation of one or more of the MPC protocols.

In [30], a parallel machine uses quantum randomness and self-modification to emulate the execution of a Universal Turing machine so that the firing patterns of the parallel machine's active elements are random to an outside observer. A procedure, described in [30], requires a novel neuromorphic hardware architecture to effectively implement a quantum random blackbox.

9. Summary

Malware can subvert the purpose of a register machine program, by changing only one address in one instruction. We built an SVM software tool (coded in `ANSI C`) that implements a Turing complete, stable virtual machine. Our SVM tool hides the operands and opcodes in unprotected memory with a complexity that exceeds FHE, and the tool's procedures are feasible to implement in current processors. A red team should test under what conditions our cryptographically stable virtual machine is resistant to sabotage.

Acknowledgments

I am deeply grateful to Lawrence Reeves, and the anonymous peer reviewers for their helpful comments.

References

[1] Nwokedi Idika, Aditya P. Mathur. "A Survey of Malware Detection Techniques." Technical Report. Dept. of Computer Science. Purdue Univ., Feb. 2007.

[2] Eric Filiol. Computer Viruses: From Theory to Applications. Springer, 2005.

[3] Stephen Cook. "The P versus NP Problem." Clay Math Institute, 2000.

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

[5] John Hennessy, David Patterson. Computer Architecture. 5th Ed., Morgan Kaufmann, 2012.

[6] Harold Abelson, Gerald Sussman. Structure and Interpretation of Computer Programs. 2nd Edition, MIT Press, 491–610, 1996.

[7] Michael Stephen Fiske. "The Active Element Machine." Proceedings of Computational Intelligence: 391, Springer, 69–96, 2011.

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

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

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

[11] James R. Munkres. Topology. Prentice-Hall, 1975.

[12] Leigh Van Valen. "An Evolutionary Law." 1–30, 1973.

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

[14] Brian Kernighan, Dennis Ritchie. The C Programming Language. Prentice-Hall, 1988.

[15] Michael Stephen Fiske. "Toward a Mathematical Understanding of the Malware Problem." Proceedings of the 53rd Hawaii International Conference on System Sciences. Wailea, Hawaii, Jan. 7-10, 2020.

[16] Richard W. Hamming. "Error Detecting and Error Correcting Codes." Bell Technical Journal. 29(2), 147–160, April 1950.

[17] Miguel Herrero-Collantes, Juan C. Garcia-Escartin. "Quantum random number generators." Reviews of Modern Physics. 89(1), 015004, APS 2017.

[18] Ronald A. Fisher, Frank Yates. "Statistical tables for biological, agricultural and medical research." London: Oliver & Boyd. 26–27, 1938.

[19] Richard Durstenfeld. "Algorithm 235: Random Permutation." Communications of the ACM. 7(7), 420, July 1964.

[20] Michael Wahl, et. al. "An ultrafast quantum random number generator based on photon arrival time measurements." Applied Physics Letters. 98, 171105, 2011.

[21] Zhijie J. Shi, Ruby B. Lee. "Implementation Complexity of Bit Permutation Instructions." Proc. of Asimolar on Signals, Systems, and Computers. 879–886, 2003.

[22] Ivan Sutherland, Bob Sproull, David Harris. "Logical Effort: Designing Fast CMOS Circuits." 1999.

[23] Mark Horowitz, et al. "Fanout-of-4 Inverter Delay Metric." CiteSeerX 10.1.1.68.831, 1998.

[24] Gaurav S. Kc, et. al. "Countering Code-Injection Attacks With Instruction-Set Randomization." CCS 2003. ACM, October 27-30, 2003.

[25] Craig Gentry. A Fully Homomorphic Encryption Scheme. Ph.D. Thesis, Stanford University, 2009.

[26] Ilaria Chillotti, et. al. "TFHE: Fast Fully Homomorphic Encryption over the Torus." Journal of Cryptology. 33, 34–91, 2020.

[27] Ravital Solomon. "An Intro to Fully Homomorphic Encryption for Engineers." July 2, 2020. Online at `https://ravital.github.io/about`.

[28] Andrew Yao. "Protocols for Secure Computations." 23rd Annual Symposium on Foundations of Computer Science. IEEE Computer Society, 160–164, 1982.

[29] David Evans, et. al. A Pragmatic Introduction to Secure Multi-Party Computation. NOW, 2021.

[30] Michael Stephen Fiske. "Turing Incomputable Computation." Turing-100 Proceedings. EasyChair 10, 66–91, 2012.