Halting problem proofs refuted on the basis of software engineering ?

This is an explanation of a possible new insight into the halting problem provided in the language of software engineering. Technical computer science terms are explained using software engineering terms. No knowledge of the halting problem is required.

It is based on fully operational software executed in the x86utm operating system. The x86utm operating system (based on an excellent open source x86 emulator) was created to study the details of the halting problem proof counter-examples at the much higher level of abstraction of C/x86.

```
typedef void (*ptr)();
int H(ptr p, ptr i); // simulating halt decider

void P(ptr x)
{
  int Halt_Status = H(x, x);
  if (Halt_Status)
    HERE: goto HERE;
  return;
}

int main()
{
  Output("Input_Halts = ", H(P, P));
}
```

When simulating halt decider H(P,P) simulates its input it can see that:
(1) Function H() is called from P().
(2) With the same arguments to H().
(3) With no instructions in P preceding its invocation of H(P,P) that could escape repeated simulations.

This is the same criteria used for infinite recursion detection that has been adapted so that it does not need static local memory to see that the same function has been called with the same arguments twice in sequence with no conditional-branch escape.

Because H knows its own machine address H need not see P call H(P,P) more than once because H already knows that it was called with these same arguments. This eliminates the need for H to have static local memory that communicates between different invocations of itself.

The above shows that the simulated P cannot possibly (reachs it “return” instruction and) terminate normally. H(P,P) simulates its input then P calls H(P,P) to simulate itself again. When H sees that this otherwise infinitely nested simulation would never end it aborts its simulation of P and rejects P as non-halting.

> In computability theory, the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever. Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.

> For any program H that might determine if programs halt, a "pathological" program P, called with some input, can pass its own source and its input to H and then specifically do the opposite of what H predicts P will do. **No H can exist that handles this case.**
> https://en.wikipedia.org/wiki/Halting_problem

H and P implement the exact pathological relationship to each other as described above. Because H(P,P) does handle this case the above halting problem undecidable input template has been refuted.

When this halt deciding principle understood to be correct:
A halt decider must compute the mapping from its inputs to an accept or reject state on the basis of the actual behavior that is actually specified by these inputs.

Within the common knowledge that the correct simulation of a program (or TM description) accurately measures the actual behavior of this program:

Then (by logical necessity) this correctly implements the halting deciding principle:
Every simulating halt decider that correctly simulates its input until it correctly predicts that this simulated input would never terminate normally, correctly rejects this input as non-halting.

H may or may not be an actual computable function. In any case H should at least apply to the **Termination analysis**. It really seems that H is a **Pure function** thus implements a **Computable function** **Thus H is Turing computable.**

A halt decider must compute the mapping from its inputs to an accept or reject state on the basis of the actual behavior that is actually specified by these inputs.

It is common knowledge that a correct simulation of a program is a correct measure of the behavior of this program. The concept of a Universal Turing Machine (UTM) is invalidated unless it is accepted that the correct simulation of a machine description is computationally equivalent to the underlying computation.

Example 03 shows the details of the execution trace of H(P,P) proving that this input would never reach its "C:"return" or x86:"ret" instruction.

computation that halts … the Turing machine will halt whenever it enters a final state. (Linz:1990:234)

Linz, Peter 1990. An Introduction to Formal Languages and Automata. Lexington/Toronto: D. C. Heath and Company. (317-320)

Example 01: H0 correctly determines that Infinite_Loop() never halts

```
void Infinite_Loop()
{
  HERE: goto HERE;
}

int main()
{
  Output("Input_Halts = ", H0((u32)Infinite_Loop));
}

_Infinite_Loop()
[00001102](01)  55         push ebp
[00001103](02)  8bec       mov ebp,esp
[00001105](02)  ebfe       jmp 00001105
[00001107](01)  5d         pop ebp
[00001108](01)  c3         ret
Size in bytes:(0007) [00001108]

_main()
[00001192](01)  55         push ebp
[00001193](02)  8bec       mov ebp,esp
[00001195](05)  6802110000 push 00001102
[0000119a](05)  e8d3fbffff call 00000d72
[0000119f](03)  83c404     add esp,+04
[000011a2](01)  50         push eax
[000011a3](05)  68a3040000 push 000004a3
[000011a8](05)  e845f3ffff call 000004f2
[000011ad](03)  83c408     add esp,+08
[000011b0](02)  33c0       xor eax,eax
[000011b2](01)  5d         pop ebp
[000011b3](01)  c3         ret
Size in bytes:(0034) [000011b3]

 machine   stack     stack     machine    assembly
 address   address   data      code       language
 ========  ========  ========  =========  =============
[00001192][00101ef8][00000000] 55         push ebp
[00001193][00101ef8][00000000] 8bec       mov ebp,esp
[00001195][00101ef4][00001102] 6802110000 push 00001102
[0000119a][00101ef0][0000119f] e8d3fbffff call 00000d72

H0: Begin Simulation   Execution Trace Stored at:211fac
[00001102][00211f9c][00211fa0] 55         push ebp
[00001103][00211f9c][00211fa0] 8bec       mov ebp,esp
[00001105][00211f9c][00211fa0] ebfe       jmp 00001105
[00001105][00211f9c][00211fa0] ebfe       jmp 00001105
H0: Infinite Loop Detected Simulation Stopped

 if (current->Simplified_Opcode == JMP)           // JMP
  if (current->Decode_Target <= current->Address) // upward
   if (traced->Address == current->Decode_Target) // to this address
    if (Conditional_Branch_Count == 0)            // no escape
     return 1;

[0000119f][00101ef8][00000000] 83c404     add esp,+04
[000011a2][00101ef4][00000000] 50         push eax
[000011a3][00101ef0][000004a3] 68a3040000 push 000004a3
[000011a8][00101ef0][000004a3] e845f3ffff call 000004f2
Input_Halts = 0
[000011ad][00101ef8][00000000] 83c408     add esp,+08
[000011b0][00101ef8][00000000] 33c0       xor eax,eax
[000011b2][00101efc][00100000] 5d         pop ebp
[000011b3][00101f00][00000004] c3         ret
Number of Instructions Executed(554) == 8 Pages
```

Example 02: H correctly determines that Infinite_Recursion() never halts

```
void Infinite_Recursion(int N)
{
  Infinite_Recursion(N);
}

int main()
{
  Output("Input_Halts = ", H((u32)Infinite_Recursion, 0x777));
}

_Infinite_Recursion()
[000010f2](01)  55         push ebp
[000010f3](02)  8bec       mov ebp,esp
[000010f5](03)  8b4508     mov eax,[ebp+08]
[000010f8](01)  50         push eax
[000010f9](05)  e8f4ffffff call 000010f2
[000010fe](03)  83c404     add esp,+04
[00001101](01)  5d         pop ebp
[00001102](01)  c3         ret
Size in bytes:(0017) [00001102]

_main()
[000011b2](01)  55         push ebp
[000011b3](02)  8bec       mov ebp,esp
[000011b5](05)  6877070000 push 00000777
[000011ba](05)  68f2100000 push 000010f2
[000011bf](05)  e8aefdffff call 00000f72
[000011c4](03)  83c408     add esp,+08
[000011c7](01)  50         push eax
[000011c8](05)  68a3040000 push 000004a3
[000011cd](05)  e820f3ffff call 000004f2
[000011d2](03)  83c408     add esp,+08
[000011d5](02)  33c0       xor eax,eax
[000011d7](01)  5d         pop ebp
[000011d8](01)  c3         ret
Size in bytes:(0039) [000011d8]

 machine   stack     stack     machine    assembly
 address   address   data      code       language
 ========  ========  ========  =========  =============
[000011b2][00101f39][00000000] 55         push ebp
[000011b3][00101f39][00000000] 8bec       mov ebp,esp
[000011b5][00101f35][00000777] 6877070000 push 00000777
[000011ba][00101f31][000010f2] 68f2100000 push 000010f2
[000011bf][00101f2d][000011c4] e8aefdffff call 00000f72

H: Begin Simulation   Execution Trace Stored at:111fe5
[000010f2][00111fd1][00111fd5] 55         push ebp
[000010f3][00111fd1][00111fd5] 8bec       mov ebp,esp
[000010f5][00111fd1][00111fd5] 8b4508     mov eax,[ebp+08]
[000010f8][00111fcd][00000777] 50         push eax      // push 0x777
[000010f9][00111fc9][000010fe] e8f4ffffff call 000010f2 // call Infinite_Recursion
[000010f2][00111fc5][00111fd1] 55         push ebp
[000010f3][00111fc5][00111fd1] 8bec       mov ebp,esp
[000010f5][00111fc5][00111fd1] 8b4508     mov eax,[ebp+08]
[000010f8][00111fc1][00000777] 50         push eax      // push 0x777
[000010f9][00111fbd][000010fe] e8f4ffffff call 000010f2 // call Infinite_Recursion
H: Infinite Recursion Detected Simulation Stopped
```

If the execution trace of function X() called by function Y() shows:

(1) Function P() is called twice in sequence from the same machine address of Y().

(2) With the same parameters to X().

(3) With no control flow instructions between the invocation of X() and the call to Y() from X().

```
[000011c4][00101f39][00000000] 83c408     add esp,+08
[000011c7][00101f35][00000000] 50         push eax
[000011c8][00101f31][000004a3] 68a3040000 push 000004a3
[000011cd][00101f31][000004a3] e820f3ffff call 000004f2
Input_Halts = 0
[000011d2][00101f39][00000000] 83c408     add esp,+08
[000011d5][00101f39][00000000] 33c0       xor eax,eax
[000011d7][00101f3d][00000018] 5d         pop ebp
[000011d8][00101f41][00000000] c3         ret
Number of Instructions Executed(1118) == 17 Pages
```

Example 03: H(P,P) correctly determines that its input never halts

```
void P(ptr x)
{
  int Halt_Status = H(x, x);
  if (Halt_Status)
    HERE: goto HERE;
  return;
}

int main()
{
  Output("Input_Halts = ",  H(P, P));
}
```

From a purely software engineering perspective (anchored in the semantics of the x86 language) it is proven that H(P,P) correctly predicts that its correct and complete x86 emulation of its input would never reach the "ret" instruction (final state) of this input.

```
_P()
[000013c6](01)  55         push ebp              // Save Base Pointer register onto the stack
[000013c7](02)  8bec       mov ebp,esp           // Load Base Pointer with Stack Pointer
[000013c9](01)  51         push ecx              // Save the value of ecx on the stack
[000013ca](03)  8b4508     mov eax,[ebp+08]      // Load eax with argument to P
[000013cd](01)  50         push eax              // push 2nd argument to H onto the stack
[000013ce](03)  8b4d08     mov ecx,[ebp+08]      // Load ecx with with argument to P
[000013d1](01)  51         push ecx              // push 1st argument to H onto the stack
[000013d2](05)  e82ffdffff call 00001106         // push return address on the stack; call simulated H
[000013d7](03)  83c408     add esp,+08           // remove call arguments from stack
[000013da](03)  8945fc     mov [ebp-04],eax      // load Halt_Status with return value from H
[000013dd](04)  837dfc00   cmp dword [ebp-04],+00 // compare Halt_Status to 0
[000013e1](02)  7402       jz 000013e5           // if Halt_Status == 0 goto 000013e5
[000013e3](02)  ebfe       jmp 000013e3          // goto 13e3
[000013e5](02)  8be5       mov esp,ebp           // Load Stack Pointer with Base Pointer
[000013e7](01)  5d         pop ebp               // Restore Base Pointer value from stack
[000013e8](01)  c3         ret                   // return to caller
Size in bytes:(0035) [000013e8]

_main()
[000013f6](01)  55         push ebp      // Save Base Pointer register onto the stack
[000013f7](02)  8bec       mov ebp,esp   // Load Base Pointer with Stack Pointer
[000013f9](05)  68c6130000 push 000013c6 // Push P (2nd argument to H) onto the stack
[000013fe](05)  68c6130000 push 000013c6 // Push P (1nd argument to H) onto the stack
[00001403](05)  e8fefcffff call 00001106 // push return address onto the stack and call executed H
[00001408](03)  83c408     add esp,+08   // remove call arguments from stack frame
[0000140b](01)  50         push eax      // Push return value from H onto the stack
[0000140c](05)  6837050000 push 00000537 // Push address of "Input_Halts = " onto the stack
[00001411](05)  e870f1ffff call 00000586 // call Output with its pushed arguments.
[00001416](03)  83c408     add esp,+08   // remove call arguments from stack frame
[00001419](02)  33c0       xor eax,eax   // set eax to 0
[0000141b](01)  5d         pop ebp       // Restore Base Pointer register from stack
[0000141c](01)  c3         ret           // return to 0 operating system
Size in bytes:(0039) [0000141c]

 machine   stack     stack     machine    assembly
 address   address   data      code       language
 ========  ========  ========  =========  =============
[000013f6][0010235f][00000000] 55         push ebp
[000013f7][0010235f][00000000] 8bec       mov ebp,esp
[000013f9][0010235b][000013c6] 68c6130000 push 000013c6 // Push P (2nd argument to H) onto the stack
[000013fe][00102357][000013c6] 68c6130000 push 000013c6 // Push P (1nd argument to H) onto the stack
[00001403][00102353][00001408] e8fefcffff call 00001106 // push return address; call executed H

H: Begin Simulation   Execution Trace Stored at:11240b
Address_of_H:1106
[000013c6][001123f7][001123fb] 55         push ebp
[000013c7][001123f7][001123fb] 8bec       mov ebp,esp
[000013c9][001123f3][001023c7] 51         push ecx           // Save the value of ecx on the stack
[000013ca][001123f3][001023c7] 8b4508     mov eax,[ebp+08]   // Load eax with argument to P
[000013cd][001123ef][000013c6] 50         push eax           // push 2nd argument to H onto the stack
[000013ce][001123ef][000013c6] 8b4d08     mov ecx,[ebp+08]   // Load ecx with with argument to P
[000013d1][001123eb][000013c6] 51         push ecx           // push 1st argument to H onto the stack
[000013d2][001123e7][000013d7] e82ffdffff call 00001106      // push return address; call simulated H
H: Infinitely Recursive Simulation Detected Simulation Stopped

[00001408][0010235f][00000000] 83c408     add esp,+08
[0000140b][0010235b][00000000] 50         push eax      // Push return value from H onto the stack
[0000140c][00102357][00000537] 6837050000 push 00000537 // Push address of "Input_Halts = " onto stack
[00001411][00102357][00000537] e870f1ffff call 00000586 // call Output with its pushed arguments
Input_Halts = 0
[00001416][0010235f][00000000] 83c408     add esp,+08
[00001419][0010235f][00000000] 33c0       xor eax,eax   // set eax to 0
[0000141b][00102363][00000018] 5d         pop ebp
[0000141c][00102367][00000000] c3         ret           // return to 0 operating system
Number of Instructions Executed(987) == 15 Pages
```

Halt Decider source-code

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <time.h>
#pragma warning (disable: 4717)
//#define OUTPUT_SIMULATED_LINE

#define u8  uint8_t
#define u32 uint32_t
#define u16 uint16_t

#define  s8 int8_t
#define s16 int16_t
#define s32 int32_t
typedef void (*ptr)();


typedef struct x86_Registers
{
  u32  EIP;
  u32  EAX;
  u32  EBX;
  u32  ECX;
  u32  EDX;
  u32  ESI;
  u32  EDI;
  u32  EBP;
  u32  ESP;
  u32  EFLG;
  u16  CS;
  u16  SS;
  u16  DS;
  u16  ES;
  u16  FS;
  u16  GS;
} Registers;


#define   JMP 0xEB // Simplifed OpCode for all forms of JMP
#define  CALL 0xE8 // Simplifed OpCode for all forms of CALL
#define   JCC 0x7F // Simplifed OpCode for all forms of Jump on Condition
#define   RET 0xC3 // Simplifed OpCode for all forms of Return
#define  PUSH 0x68 // Simplifed OpCode for all forms of PUSH
#define OTHER 0xFF // Not a Control Flow Insrtuction
#define   HLT 0xF4 // Conventional OpCode for Halt


typedef struct Decoded
{
  u32 Address;
  u32 ESP;          // Current value of ESP
  u32 TOS;          // Current value of Top of Stack
  u32 NumBytes;
  u32 Simplified_Opcode;
  u32 Decode_Target;
} Decoded_Line_Of_Code;


u8 BEGIN[]    = "BEGIN STATIC DATA"; // Required to force allocation
u32 Heap_PTR = 0x11111111;  // forces memory allocation
u32 Heap_END = 0x22222222;  // forces memory allocation
u8 END[]     = "END STATIC DATA";     // Required to force allocation
```

```
// Empty Stub Functions of Virtual Machine Instructions
// x86utm operating system calls
void OutputString(char* S) {}
void Output(char* S, u32 N) {}
u32* Allocate(u32 size) { return 0; }
void SaveState(Registers* state) {}
void LoadState(Registers* state) {}
u32  DebugStep(Registers* master_state,
               Registers* slave_state, Decoded_Line_Of_Code* decoded) { return 0; }
void PushBack(u32 stdvector, u32 data_ptr, u32 size_in_bytes) {}
u32  StackPush(u32* S, u32 M) { return 0; }
u32  get_code_end(u32 EIP){ return 0; }


u32 Infinite_Loop_Needs_To_Be_Aborted_Trace
    (Decoded_Line_Of_Code* execution_trace, Decoded_Line_Of_Code *current)
{
  Decoded_Line_Of_Code *traced;
  u32 Conditional_Branch_Count = 0;

  u32* ptr = (u32*)execution_trace;  // 2021-04-06
  u32 size = ptr[-1];                // 2021-04-06
  u32 next2last = (size/sizeof(Decoded_Line_Of_Code)) -2;
  for (s32 N = next2last; N >= 0; N--)
  {
    traced = &execution_trace[N];
    if (traced->Simplified_Opcode == JCC)                    // JCC
      Conditional_Branch_Count++;

    if (current->Simplified_Opcode == JMP)                   // JMP
      if (current->Decode_Target <= current->Address)        // upward
        if (traced->Address == current->Decode_Target)       // to this address
          if (Conditional_Branch_Count == 0)                 // no escape
            return 1;
  }
  return 0;
}

u32 Infinite_Recursion_Needs_To_Be_Aborted_Trace
    (Decoded_Line_Of_Code* execution_trace, Decoded_Line_Of_Code *current)
{
  Decoded_Line_Of_Code *traced;
  u32 Conditional_Branch_Count = 0;

  u32* ptr = (u32*)execution_trace;  // 2021-04-06
  u32 size = ptr[-1];                // 2021-04-06
  u32 next2last = (size/sizeof(Decoded_Line_Of_Code)) -2;
  for (s32 N = next2last; N >= 0; N--)
  {
    traced = &execution_trace[N];
    if (traced->Simplified_Opcode == JCC)                    // JCC
      Conditional_Branch_Count++;

    if (current->Simplified_Opcode == CALL)
      if (current->Simplified_Opcode == traced->Simplified_Opcode)  // CALL
        if (current->Address == traced->Address)             // from same address
          if (current->Decode_Target == traced->Decode_Target)// to Same Function
            if (Conditional_Branch_Count == 0)               // no escape
              return 2;
  }
  return 0;
}
```

```
u32 Infinite_Simulation_Needs_To_Be_Aborted_Trace
    (Decoded_Line_Of_Code* execution_trace,
     Decoded_Line_Of_Code *current, u32 P, u32 I)
{
  Decoded_Line_Of_Code *traced;
  u32 Count_PUSH_Instructions  = 0;
  u32 Num_PUSH_Matched         = 0;
  u32 Conditional_Branch_Count = 0;

  u32* ptr = (u32*)execution_trace;  // 2021-04-06
  u32 size = ptr[-1];                // 2021-04-06
  u32 next2last = (size/sizeof(Decoded_Line_Of_Code)) -2;
  for (s32 N = next2last; N >= 0; N--)
  {
    traced = &execution_trace[N];
    if (traced->Simplified_Opcode == JCC)                     // JCC
      Conditional_Branch_Count++;

    if (traced->Simplified_Opcode == PUSH)                    // PUSH
      Count_PUSH_Instructions++;

    if (traced->Simplified_Opcode == PUSH &&
        traced->Decode_Target == P && Count_PUSH_Instructions == 1)
      Num_PUSH_Matched++;

    if (traced->Simplified_Opcode == PUSH &&
        traced->Decode_Target == I && Count_PUSH_Instructions == 2)
      Num_PUSH_Matched++;

    if (Num_PUSH_Matched == 2 && N == 0 && Conditional_Branch_Count == 0)
      return 3;
  }
  return 0;
}


u32 Needs_To_Be_Aborted(Decoded_Line_Of_Code* execution_trace,
                        u32 Address_of_H, u32 P, u32 I)
{
  u32 Aborted = 0;
  u32* ptr = (u32*)execution_trace;  // 2021-04-06
  u32 size = ptr[-1];                // 2021-04-06
//Output("Needs_To_Be_Aborted(size):", size);
  u32 last = (size / sizeof(Decoded_Line_Of_Code)) - 1;
  Decoded_Line_Of_Code* current = &execution_trace[last];

  if (current->Simplified_Opcode == CALL)
  {
    if (current->Decode_Target == Address_of_H)
      Aborted = Infinite_Simulation_Needs_To_Be_Aborted_Trace
                (execution_trace, current, P, I);
    else
      Aborted = Infinite_Recursion_Needs_To_Be_Aborted_Trace
                (execution_trace, current);
  }
  else if (current->Simplified_Opcode == JMP)
    Aborted = Infinite_Loop_Needs_To_Be_Aborted_Trace(execution_trace, current);
  return Aborted;
}
```

```
//
// This is called every time the a line ocf x86 code is emulated
//
u32 Decide_Halting(char*                   Halt_Decider_Name,
                   u32*                    execution_trace,
                   Decoded_Line_Of_Code**  decoded,
                   u32                     code_end,
                   Registers**             master_state,
                   Registers**             slave_state,
                   u32**                   slave_stack,
                   u32                     Address_of_H, u32 P, u32 I)
{
  u32 Aborted = 0;
  while (Aborted == 0)
  {
    u32 EIP = (*slave_state)->EIP; // Save EIP of instruction to be executed
    DebugStep(*master_state, *slave_state, *decoded); // Execute this instruction
    if (EIP == code_end)           // last instruction of P "ret"
      return 1;                    // input has halted

#ifdef OUTPUT_SIMULATED_LINE
    Output_Decoded((u32)*decoded);
#endif
//  When we are not recursively simulatng H we don't need this is statement
//  if (EIP > Last_Address_Of_Operating_System())  // Don't examine any OS code
    PushBack(*execution_trace, (u32)*decoded, sizeof(Decoded_Line_Of_Code));
    Aborted = Needs_To_Be_Aborted((Decoded_Line_Of_Code*)*execution_trace,
                                   Address_of_H, P, I);
  }
  if (Aborted) // 2021-01-26 Must be aborted
  {
    OutputString(Halt_Decider_Name);
    if (Aborted == 1)
      OutputString("Infinite Loop Detected Simulation Stopped\n\n");
    if (Aborted == 2)
      OutputString("Infinite Recursion Detected Simulation Stopped\n\n");
    if (Aborted == 3)
      OutputString("Infinitely Recursive Simulation Detected "
                   "Simulation Stopped\n\n");
    return 0;
  }
  return 1;         // 2021-01-26 Need not be aborted
}


// This only works with ONE PARAMETER to the called function
void Init_slave_state(u32 P, u32 I, u32 End_Of_Code,
                      Registers* slave_state, u32* slave_stack)
{
  u32 Top_of_Stack;
  u32 Capacity;
  u32 Size;

  Top_of_Stack = StackPush(slave_stack, I);   // Data for Function to invoke
  Top_of_Stack = StackPush(slave_stack, End_Of_Code); // Return Address in Halts()

  SaveState(slave_state);     // Based on this point in execution
  Capacity = slave_stack[-2];
  Size = slave_stack[-1];

  slave_state->EIP = P;       // Function to invoke
  slave_state->ESP = Top_of_Stack;
  slave_state->EBP = Top_of_Stack;
}
```

```
u32 H(ptr P, ptr I)
{
HERE:
  u32 End_Of_Code;
  u32 Address_of_H;              // 2022-06-17
  u32 code_end                  = get_code_end((u32)P);
  Decoded_Line_Of_Code *decoded = (Decoded_Line_Of_Code*)
                                   Allocate(sizeof(Decoded_Line_Of_Code));
  Registers*  master_state      = (Registers*) Allocate(sizeof(Registers));
  Registers*  slave_state       = (Registers*) Allocate(sizeof(Registers));
  u32*        slave_stack       = Allocate(0x10000); // 64k;
  u32  execution_trace = (u32)Allocate(sizeof(Decoded_Line_Of_Code) * 10000);
                                  // 10000 lines of x86 code
  __asm lea eax, HERE             // 2022-06-18
  __asm sub eax, 6                // 2022-06-18
  __asm mov Address_of_H, eax     // 2022-06-18
  __asm mov eax, END_OF_CODE
  __asm mov End_Of_Code, eax

  Init_slave_state((u32)P, (u32)I, End_Of_Code, slave_state, slave_stack);
  Output("\nH: Begin Simulation   Execution Trace Stored at:", execution_trace);
  Output("Address_of_H:", Address_of_H); // 2022-06-11
  if (Decide_Halting("H: ", &execution_trace, &decoded, code_end, &master_state,
                     &slave_state, &slave_stack, Address_of_H, (u32)P, (u32)I))
      goto END_OF_CODE;
  return 0;  // Does not halt
END_OF_CODE:
  OutputString("H: End Simulation   Input Terminated Normally\n\n");
  return 1; // Input has normally terminated
}


// Dummy Place holder needed to know where
// the x86utm operating system is located.
// THIS FUNCTION MAY BE OBSOLETE
u32 Halts(u32 P, u32 I)
{
  return 0;
}


void P(ptr x)
{
  int Halt_Status = H(x, x);
  if (Halt_Status)
    HERE: goto HERE;
  return;
}

int main()
{
  Output("Input_Halts = ", H(P, P));
}
```

Appendix (Simulating halt decider applied to Peter Linz proof)

The following is the same idea a shown above this time it is applied to the Peter Linz Halting Problem proof. It can only be undertood within the context of this proof.

A simulating halt decider (SHD) computes the mapping from its inputs to its own final states on the basis of the behavior of its correctly simulated input.

All of the conventional halting problem counter-example inputs are simply rejected by a simulating halt decider as non-halting because they fail to meet the Linz definition of halting:

computation that halts … the Turing machine will halt whenever it enters a final state. (Linz:1990:234)

USENET comp.theory: On 4/11/2022 3:19 PM, Malcolm McLean wrote:
> PO's idea is to have a simulator with an infinite cycle detector.
> You would achieve this by modifying a UTM, so describing it as
> a "modified UTM", or "acts like a UTM until it detects an infinite
> cycle", is reasonable. And such a machine is a fairly powerful
> halt decider. Even if the infinite cycle detector isn't very
> sophisticated, it will still catch a large subset of non-halting
> machines.

The following simplifies the syntax for the definition of the Linz Turing machine Ĥ.
There is no need for the infinite loop after H.qy because it is never reached. The halting criteria has been adapted so that it applies to a simulating halt decider (SHD).

Ĥ.q_0 ⟨Ĥ⟩ ⊢* H ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qy
If the correctly simulated input ⟨Ĥ⟩ ⟨Ĥ⟩ to H would reach its own final state of ⟨Ĥ.qy⟩ or ⟨Ĥ.qn⟩.

Ĥ.q_0 ⟨Ĥ⟩ ⊢* H ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qn
If the correctly simulated input ⟨Ĥ⟩ ⟨Ĥ⟩ to H would never reach its own final state of ⟨Ĥ.qy⟩ or ⟨Ĥ.qn⟩.

When Ĥ is applied to ⟨Ĥ⟩ // subscripts indicate unique finite strings
Ĥ copies its input ⟨$\hat{H}_0$⟩ to ⟨$\hat{H}_1$⟩ then H simulates ⟨$\hat{H}_0$⟩ ⟨$\hat{H}_1$⟩

Then these steps would keep repeating: (unless their simulation is aborted)
$\hat{H}_0$ copies its input ⟨$\hat{H}_1$⟩ to ⟨$\hat{H}_2$⟩ then H_0 simulates ⟨$\hat{H}_1$⟩ ⟨$\hat{H}_2$⟩
$\hat{H}_1$ copies its input ⟨$\hat{H}_2$⟩ to ⟨$\hat{H}_3$⟩ then H_1 simulates ⟨$\hat{H}_2$⟩ ⟨$\hat{H}_3$⟩
$\hat{H}_2$ copies its input ⟨$\hat{H}_3$⟩ to ⟨$\hat{H}_4$⟩ then H_2 simulates ⟨$\hat{H}_3$⟩ ⟨$\hat{H}_4$⟩...

Since we can see that the simulated input: ⟨$\hat{H}_0$⟩ to H would never reach its own final state of ⟨$\hat{H}_0$.qy⟩ or ⟨$\hat{H}_0$.qn⟩ we know that it is non-halting.

Linz, Peter 1990. An Introduction to Formal Languages and Automata. Lexington/Toronto: D. C. Heath and Company. (317-320)

Infinite recursion / infinitely recursive emulation detection criteria

```
int H(ptr p, ptr i)
{
  p(i);
}

void P(ptr x)
{
  H(x, x);
  return;
}

int main()
{
  H(P,P);
}
```

If the execution trace of function P() called by function H() shows:
(1) Function H() is called twice in sequence from the same machine address of P().
(2) With the same parameters to H().
(3) With no control flow instructions between the invocation of P() and the call to H() from P().

Then the function call from P() to H() is infinitely recursive.
The exact same pattern applies when H() simulates its input with an x86 emulator.

When H is an infinite recursion detector it simply matches the above criteria in its execution trace of P, aborts its simulation of its input and reports that its simulated input would never reach its "return" instruction.

To avoid using static local memory for its stored execution trace H must know its own address and see itself called from P with the same arguments that it was called with.

https://www.liarparadox.org/2022_07_22.zip
This is the complete system that compiles under:

Microsoft Visual Studio Community 2017
https://visualstudio.microsoft.com/vs/older-downloads/

It has not been recently compiled under UBUNTU

If a simulating halt decider continues to correctly simulate its input until it correctly matches a non-halting behavior pattern then this SHD is necessarily correct when it aborts its simulation and reports non-halting.

Halt Decider

```
07/22/2022  07:05 AM   10,390 Halt7.c
```

x86utm operating system

```
07/22/2022  07:08 AM   72,499 x86utm.cpp
07/01/2022  03:40 PM   32,931 Read_COFF_Object.h
```

x86 emulator source-code

```
10/04/2020  07:44 PM   17,240 api.c
05/03/2021  02:33 PM   58,872 decode.c
06/28/2020  05:22 PM   20,495 mem.c
06/30/2020  12:06 AM  141,330 ops.c
06/27/2022  11:15 AM   63,704 ops2.c
06/30/2020  12:09 AM   73,787 prim_ops.c
01/28/2020  06:19 AM    5,542 decode.h
02/17/2020  08:56 PM   18,558 getopt.h
01/28/2020  06:19 AM    1,859 mem.h
01/28/2020  06:19 AM    1,954 ops.h
01/28/2020  06:19 AM    6,133 prim_ops.h
02/09/2021  10:40 PM   19,596 x86emu.h
10/04/2020  07:34 PM    4,996 x86emu_int.h
```

Compiles into x86utm.exe