# Halting problem undecidability and infinitely nested simulation (V2)

The halting theorem counter-examples present infinitely nested simulation (non-halting) behavior to every simulating halt decider. This paper has been rewritten to be more compelling and more concise.

The pathological self-reference of the conventional halting problem proof counter-examples is overcome. The halt status of these examples is correctly determined. A simulating halt decider remains in pure simulation mode until after it determines that its input will never reach its final state. This eliminates the conventional feedback loop where the behavior of the halt decider effects the behavior of its input.

The x86utm operating system was created so that the halting problem could be examined concretely in the high level language of C. H is a function written in C that analyzes the x86 machine language execution trace of other functions written in C. H recognizes simple cases of infinite recursion and infinite loops. The conventional halting problem proof counter-example template is shown to simply be an input that does not halt.

H simulates its input with an x86 emulator until it determines that its input would never halt. As soon as H recognizes that its input would never halt it stops simulating this input and returns 0. For inputs that do halt H acts exactly as if it was an x86 emulator and simply runs its input to completion and then returns 1.

> In theoretical computer science the random-access stored-program (RASP) machine model is an abstract machine used for the purposes of algorithm development and algorithm complexity theory. ...The RASP is closest of all the abstract models to the common notion of computer.
> https://en.wikipedia.org/wiki/Random-access_stored-program_machine

The C/x86 model of computation is known to be Turing equivalent on the basis that it maps to the RASP model for all computations having all of the memory that they need. As long as an C/x86 function is a pure function of its inputs the C/x86 model of computation can be relied upon as a much higher level of abstraction of the behavior of actual Turing machines.

This criteria merely relies on the fact that the UTM simulation of a machine description of a machine is computationally equivalent to the direct execution of this same machine:

**Simulating Halt Decider Theorem (Olcott 2021)(V3):**
Whenever simulating halt decider H correctly determines that it simulation of input P would never reach its final state (whether or not this simulation is aborted) then H correctly decides that P never halts.

> 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. https://en.wikipedia.org/wiki/Halting_problem

As long as the halt decider H reports the halt status of the behavior of its actual input P then H is necessarily correct no matter how P behaves in any other situation.

Because H only acts as a pure simulator of its input until after its halt status decision has been made it has no behavior that can possibly effect the behavior of its input. Because of this H screens out its own address range in every execution trace that it examines. This is why we never see any instructions of H in any execution trace after an input calls H.

**Pathological Input** to a halt decider is stipulated to mean any input that was defined to do the opposite of whatever its corresponding halt decider decides as Sipser describes:

> Now we construct a new Turing machine D with H as a subroutine.
> This new TM calls H to determine what M does when the input to M
> is its own description ⟨M⟩. Once D has determined this information,
> it does the opposite. (Sipser:1997:165)

When D is invoked with input **⟨D⟩** we have pathological self-reference when D calls H with ⟨D⟩ and does the opposite of whatever H returns.

**Does D halt on its own machine description ⟨D⟩ ?**
This question can only be correctly answered after the pathology has been removed. When a halt decider only acts as a pure simulator of its input until after its halt status decision is made there is no feedback loop of back channel communication between the halt decider and its input that can prevent a correct halt status decision. In this case the halt decider is only examining the behavior of the input and has no behavior that can effect the behavor of this input thus can ignore it own behavior.

The standard pseudo-code halting problem template "proved" that the halting problem could never be solved on the basis that neither value of true (halting) nor false (not halting) could be correctly returned form the halt decider to the confounding input.

```
procedure compute_g(i):  // (Wikipedia:Halting Problem)
  if f(i, i) == 0 then   // adapted from (Strachey, C 1965)
    return 0             // originally written in CPL
  else                   // ancestor of the BCPL, B and C
    loop forever         // programming languages
```

This problem is overcome on the basis that a simulating halt decider would abort the simulation of its input before ever returning any value to this input. It aborts the simulation of its input on the basis that its input specifies what is essentially infinite recursion (infinitely nested simulation) to any simulating halt decider.

Every input to a simulating halt decider that only stops running when its simulation is aborted unequivocally specifies a computation that never halts. When input to a simulating halt decider cannot possibly reach its final state then we know that this input never halts.

**A simulating halt decider H divides all of its input into:**
(1) Those inputs that never halt unless H aborts their simulation (never halting).
H aborts its simulation of these inputs an returns 0 for never halting.

(2) Those inputs that halt while H remains a pure simulator (halting).
H waits for its simulation of this input to complete and then returns 1 halting.

**Simulating partial halt decider H correctly decides that P(P) never halts (V1)**

The execution trace of the first 14 steps of the correct pure simulation of the input to H(P,P) conclusively proves that the x86 source code of P specifies infinitely nested simulation to every simulating halt decider H.

Whether or not H aborts the simulation of its input this input never reaches its final state at c50. This conclusively proves that the only possible correct return value for H(P,P) is 0 indicating that the pure simulation of its input never halts.

```
// Simplified Linz Ĥ (Linz:1990:319)
// Strachey(1965) CPL translated to C
void P(u32 x)
{
  if (H(x, x))
    HERE: goto HERE;
}

int main()
{
  Output("Input_Halts = ", H((u32)P, (u32)P));
}

_P()
[00000c36](01)  55          push ebp
[00000c37](02)  8bec        mov ebp,esp
[00000c39](03)  8b4508      mov eax,[ebp+08] // 2nd Param
[00000c3c](01)  50          push eax
[00000c3d](03)  8b4d08      mov ecx,[ebp+08] // 1st Param
[00000c40](01)  51          push ecx
[00000c41](05)  e820fdffff  call 00000966    // call H
[00000c46](03)  83c408      add esp,+08
[00000c49](02)  85c0        test eax,eax
[00000c4b](02)  7402        jz 00000c4f
[00000c4d](02)  ebfe        jmp 00000c4d
[00000c4f](01)  5d          pop ebp
[00000c50](01)  c3          ret
Size in bytes:(0027) [00000c50]

_main()
[00000c56](01)  55          push ebp
[00000c57](02)  8bec        mov ebp,esp
[00000c59](05)  68360c0000  push 00000c36    // push P
[00000c5e](05)  68360c0000  push 00000c36    // push P
[00000c63](05)  e8fefcffff  call 00000966    // call H(P,P)
[00000c68](03)  83c408      add esp,+08
[00000c6b](01)  50          push eax
[00000c6c](05)  6857030000  push 00000357
[00000c71](05)  e810f7ffff  call 00000386
[00000c76](03)  83c408      add esp,+08
[00000c79](02)  33c0        xor eax,eax
[00000c7b](01)  5d          pop ebp
[00000c7c](01)  c3          ret
Size in bytes:(0039) [00000c7c]

 machine   stack     stack     machine    assembly
 address   address   data      code       language
 ========  ========  ========  =========  =============
[00000c56][0010172a][00000000] 55          push ebp
[00000c57][0010172a][00000000] 8bec        mov ebp,esp
[00000c59][00101726][00000c36] 68360c0000  push 00000c36 // push P
[00000c5e][00101722][00000c36] 68360c0000  push 00000c36 // push P
[00000c63][0010171e][00000c68] e8fefcffff  call 00000966 // call H(P,P)
```

```
Begin Local Halt Decider Simulation at Machine Address:c36
[00000c36][002117ca][002117ce] 55          push ebp
[00000c37][002117ca][002117ce] 8bec        mov ebp,esp
[00000c39][002117ca][002117ce] 8b4508      mov eax,[ebp+08]
[00000c3c][002117c6][00000c36] 50          push eax       // push P
[00000c3d][002117c6][00000c36] 8b4d08      mov ecx,[ebp+08]
[00000c40][002117c2][00000c36] 51          push ecx       // push P
[00000c41][002117be][00000c46] e820fdffff  call 00000966  // call H(P,P)

[00000c36][0025c1f2][0025c1f6] 55          push ebp
[00000c37][0025c1f2][0025c1f6] 8bec        mov ebp,esp
[00000c39][0025c1f2][0025c1f6] 8b4508      mov eax,[ebp+08]
[00000c3c][0025c1ee][00000c36] 50          push eax       // push P
[00000c3d][0025c1ee][00000c36] 8b4d08      mov ecx,[ebp+08]
[00000c40][0025c1ea][00000c36] 51          push ecx       // push P
[00000c41][0025c1e6][00000c46] e820fdffff  call 00000966  // call H(P,P)
Local Halt Decider: Non halting behavior detected simulation stopped
```

We verify that H(P,P) does perform a correct pure simulation the first seven instructions of its input on the basis of the one-to-one mapping of the first seven instructions of the x86 source-code of P to the first seven steps of the listed simulation of P.

We don't need to see the hundreds of pages of the simulation of H to know that when seventh instruction of P simply calls H(P,P) again these same seven steps would be repeated in a nested simulation.

From these 14 correctly simulated steps we can see that the nested simulations never stop unless H aborts it.

Because we know that the correctly simulated input to H(P,P) never reaches its final state whether or not H(P,P) aborts the simulation of its input we know that H(P,P)==0 is correct for every simulating halt decider H.

To refute the claim that the direct execution of P(P) halts thus its simulation is incorrect H(P,P) directly executes its input instead of simulating it. The result is that the infinitely nested simulation becomes infinite recursion. In no case does the following directly executed P ever reach its final state of c50.

```
int H(u32 P, u32 I)
{
  if (!Halts(P,I)
    return 0;
  ((void(*)(int))P)(I); // call void P(I);
  return 1;
}

int main()
{
  H((u32)P, (u32)P);
}
```

## Peter Linz Ĥ applied to the Turing machine description of itself: ⟨Ĥ⟩

The following simplifies the syntax for the definition of the Linz Turing machine Ĥ, it is now a single machine with a single start state. A simulating halt decider is embedded at Ĥ.qx. It has been annotated so that it only shows Ĥ applied to ⟨Ĥ⟩, converting the variables to constants.

**Ĥ.q0 ⟨Ĥ⟩ ⊢* Ĥ.qx ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qy ∞**
**If the UTM simulation of the input to Ĥ.qx ⟨Ĥ⟩ applied to ⟨Ĥ⟩ reaches its own final state.**

**Ĥ.q0 ⟨Ĥ⟩ ⊢* Ĥ.qx ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qn**
If the pure simulation of the input to Ĥqx ⟨Ĥ⟩ ⟨Ĥ⟩ would never reach its final state (whether or not this simulation is aborted) then it is necessarily true that Ĥqx transitions to Ĥ.qn correctly.

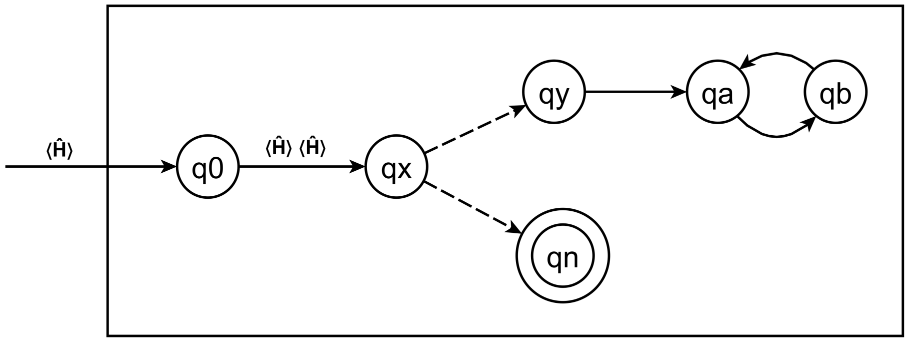

**Figure 12.3 Turing Machine Ĥ applied to ⟨Ĥ⟩**

Ĥ.q0 copies its input ⟨$Ĥ_0$⟩ to ⟨$Ĥ_1$⟩ then Ĥ.qx simulates this input $Ĥ_0$ with its input ⟨$Ĥ_1$⟩
$Ĥ_0$.q0 copies its input ⟨$Ĥ_1$⟩ to ⟨$Ĥ_2$⟩ then $Ĥ_0$.qx simulates this input $Ĥ_1$ with its input ⟨$Ĥ_2$⟩
$Ĥ_1$.q0 copies its input ⟨$Ĥ_2$⟩ to ⟨$Ĥ_3$⟩ then $Ĥ_1$.qx simulates this input $Ĥ_2$ with its input ⟨$Ĥ_3$⟩
$Ĥ_2$.q0 copies its input ⟨$Ĥ_3$⟩ to ⟨$Ĥ_4$⟩ then $Ĥ_2$.qx simulates this input $Ĥ_3$ with its input ⟨$Ĥ_4$⟩ ...

If the simulating halt decider at Ĥ.qx never aborts its simulation of its input this input never halts. If Ĥ.qx aborts its simulation of its input this input never reaches its final state and thus never halts.

If the input to Ĥ.qx ⟨Ĥ⟩ ⟨Ĥ⟩ never halts then the transition to ⊢* Ĥ.qn is necessarily correct no matter what Ĥ ⟨Ĥ⟩ does. A halt decider is only accountable for correctly deciding the halt status of its actual input.

When the original Linz H is applied to **⟨Ĥ⟩ ⟨Ĥ⟩** it sees that its input transitions to Ĥ.qn. This provides the basis for H to transition to its final state of H.qy.

When Ĥ.qx is applied to **⟨Ĥ⟩ ⟨Ĥ⟩** it sees that none of the recursive simulations of its input ever halt so it aborts the simulation of its input and transtions to its final state of Ĥ.qn.

Copyright 2016-2021 PL Olcott

**Strachey, C 1965.** An impossible program The Computer Journal, Volume 7, Issue 4, January 1965, Page 313, https://doi.org/10.1093/comjnl/7.4.313

**Linz, Peter 1990**. An Introduction to Formal Languages and Automata. Lexington/Toronto: D. C. Heath and Company. (318-320)

**Sipser, Michael 1997**. Introduction to the Theory of Computation. Boston: PWS Publishing Company (165-167)