Halting problem proofs refuted on the basis of software engineering

This is an explanation of a key 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.

To fully understand this paper a software engineer must be an expert in:
(a) The C programming language.
(b) The x86 programming language.
(c) Exactly how C translates into x86 (how C function calls are implemented in x86).
(d) The ability to recognize infinite recursion at the x86 assembly language level.

The computer science term “halting” means that a Turing Machine terminated normally reaching its last instruction known as its “final state”. This is the same idea as when a function returns to its caller as opposed to and contrast with getting stuck in an infinite loop or infinite recursion.

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](https://en.wikipedia.org/wiki/Halting_problem)

H and P implement the above specified pathological relationship to each other:

```c
void P(u32 x)
{
  if (H(x, x)) {
    HERE: goto HERE;
    return;
  }
}

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

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.

This general principle refutes conventional halting problem proofs
Every simulating halt decider that correctly simulates its input until it correctly predicts that this simulated input would never reach its final state, correctly rejects this input as non-halting.

From a purely software engineering perspective H(P,P) is required to correctly predict that its correct and complete x86 emulation of its input would never reach the "ret" instruction of this input and H must do this in a finite number of steps. (see appendix).
Appendix (three examples)

H0 correctly determines that Infinite_Loop() never halts

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

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

```asm
_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
[00001199](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**

<table>
<thead>
<tr>
<th>Address</th>
<th>Stack Data</th>
<th>Stack Code</th>
<th>Machine Address</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001102</td>
<td>00101ef8</td>
<td>00000000</td>
<td>55</td>
<td>push ebp</td>
</tr>
<tr>
<td>00001103</td>
<td>00101ef8</td>
<td>00000000</td>
<td>8bec</td>
<td>mov ebp,esp</td>
</tr>
<tr>
<td>00001105</td>
<td>00101ef4</td>
<td>00001102</td>
<td>6802110000</td>
<td>push 00001102</td>
</tr>
<tr>
<td>00001109</td>
<td>00101ef0</td>
<td>0000119f</td>
<td>e8d3fbffff</td>
<td>call 00000d72</td>
</tr>
</tbody>
</table>

**H0: Begin Simulation**

Execution Trace Stored at: 211fac

**H0: Infinite Loop Detected**

Simulation Stopped

```c
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;
```

**Input_Halts = 0**

Number of Instructions Executed(554) == 8 Pages
H correctly determines that Infinite_Recursion() never halts

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

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

H: Begin Simulation Execution Trace Stored at:111fe5

```
if (current->Simplified_Opcode == CALL)
    if (current->Address == traced->Address)
        if (Conditional_Branch_Count == 0)
            return 2;
```

Number of Instructions Executed(1118) == 17 Pages
H(P,P) correctly determines that its input never halts

```c
void P(u32 x)
{
    if (H(x, x))
    {
        HERE: goto HERE;
    }
    return;
}

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

H knows its own machine address and on this basis it can easily examine its stored execution_trace of P (see above) to determine:

(a) P is calling H with the same arguments that H was called with.
(b) No instructions in P could possibly escape this otherwise infinitely recursive emulation.
(c) H aborts its emulation of P before its call to H is emulated.

---
07/04/22 09:05:33 AM
---

Copyright 2022 PL Olcott