Conditionally Executed Malware Detection Through Emulation in ASP

By Aiden McClelland

Introduction

When using emulation as a means of malware detection, it is possible to miss potential malware because it requires specific conditions for activation that are not met by the emulated environment. In order to combat this, I have developed a system that uses Answer Set Programming to model an x86 processor and instruction set to detect if there exists a set of results from API calls that will result in malicious behavior, and report what those conditions are. This would be useful in diagnosing malware that does not activate unless certain conditions are met. The system would ideally take in a Windows x86 portable executable (PE), however the current prototype takes in an intermediate format similar to assembly language.

Background

Currently there exists a tool that will execute a program within a virtual machine, take a snapshot at each branching statement, and revert to a previous snapshot that has not been fully explored upon termination of the program, until all paths of execution through the program have been explored. In practice, this would be remarkably inefficient as virtual machine snapshot operations are costly, and exploring every single branch would scale very poorly, not to mention, branches that jump backwards in execution, such as those used in loops, will only run the loop once, which could throw off what proper execution would be. By modeling a processor using ASP, logical reasoning can be performed to reduce the space of paths through the program only to those which are possible during its actual execution, and to actively search for paths which create bad behavior. In practice, this project will be more similar to Microsoft's project to use SAT solvers to determine if an execution through a program exists in which assertions will fail.

ASP Formulation

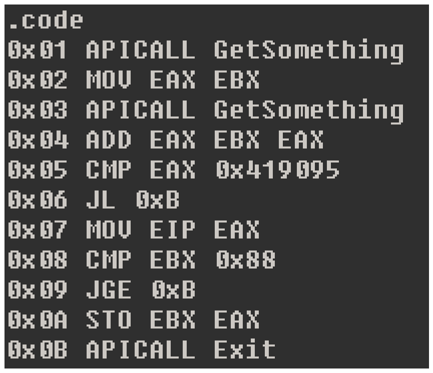
In order to facilitate the system, an x86 processor and instruction set must be modeled in ASP. To do so, each register is represented as a series of facts indicating a 1 at a specific position in the register at a specific time. The absence of such a fact indicates a zero at that position and time. With the exception of the EIP register, all of these registers needed to maintain their value between time steps unless acted upon by an ADD, SUB, MOV, etc. The EIP register needed to advance every time step, unless acted upon by a jump instruction. To do so, I made a simple incrementer circuit using rules in ASP. For the ADD, SUB, and CMP instructions, I implemented a ripple carry adder. This adder also set all of the necessary flags for the jump instructions. Finally, API calls were represented as choice rules on the EAX register, indicating a non-deterministic return value.

In order to have the system detect malware, malware signatures had to be defined, and a forbid rule was written to reject solutions that do not result in malicious behaviour. An eicar test signature was defined as if the PC ever reached the eicar instruction before an exit was called. Another signature was defined as if the program ever tried to modify its own instructions.

To determine when the EIP was pointing at a specific instruction, a separate lp file had to be generated that would convert the bitwise representation to a number.

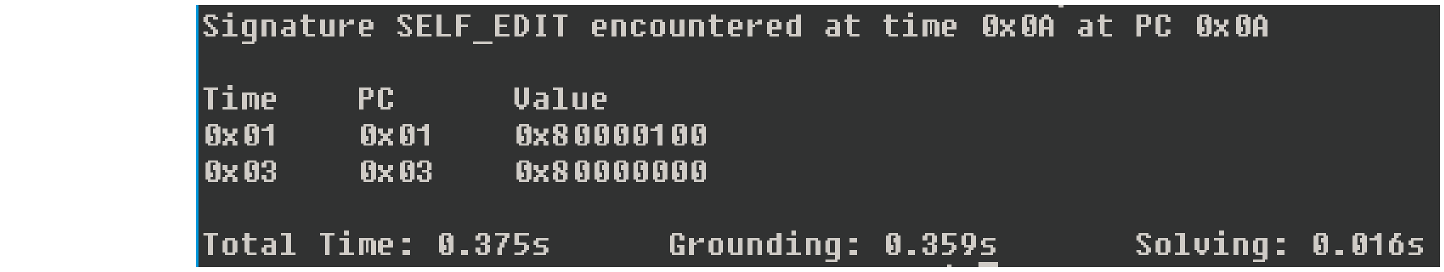
Problem Instance

Let's say, for example, that we have a program that has the potential to modify its own instructions like so:



```
.code
0x01 APICALL GetSomething
0x02 MOV EAX EBX
0x03 APICALL GetSomething
0x04 ADD EAX EBX EAX
0x05 CMP EAX 0x419095
0x06 JL 0xB
0x07 MOV EIP EAX
0x08 CMP EBX 0x88
0x09 JGE 0xB
0x0A STO EBX EAX
0x0B APICALL Exit
```

This program requires that the values returned from these API calls meet certain conditions in order to do so: `Call1 + Call2 >= 0x419095` AND `Call1 < 0x88`. To detect this potentially malicious code, the system must be able to identify 2 values that meet these conditions. As you can see below, the system identifies that if it selects these large negative values for the 2 API calls, when added together they will overflow to a value larger than `0x419095`, and `Call1` will be less than `0x88` by nature of being negative.



```
Signature SELF_EDIT encountered at time 0x0A at PC 0x0A

Time     PC       Value
0x01     0x01     0x80000100
0x03     0x03     0x80000000

Total Time: 0.375s      Grounding: 0.359s       Solving: 0.016s
```

Formulation

Formulation Notebook¶

A Sample Problem Instance¶

Problem instances take the form of an x86 PE for Windows. Since converting these files is non-trivial, I have created a pseudo-assembly language format for describing problem instances.

In [1]:

```
%%file instance-6.ins
.code
0x01 APICALL GetSomething
0x02 MOV EAX EBX
0x03 APICALL GetSomething
0x04 ADD EAX EBX EAX
0x05 CMP EAX 0x419095
0x06 JL 0xB
0x07 NOP
0x08 CMP EBX 0x88
0x09 JGE 0x01
0x0A EICAR
0x0B APICALL Exit
.data
```

```
Overwriting instance-6.ins
```

This program will only execute the eicar instruction under a few conditions: the sum of the results of the first and second API calls should be at least 0x419095, and the result of the first API call should be less than 0x88. If out formulation is successful, we should recieve results from these apicalls that reflect these conditions.

Converting the problem instance to facts¶

In order to be able to read in the problem instance, it must be converted to a list of facts that can be read by clingo. To do so, I use a python program.

In [2]:

```
%%file geninstance.py

import re

def convert_case(name):
    s1 = re.sub('(.)([A-Z][a-z]+)', r'\1_\2', name)
    return re.sub('([a-z0-9])([A-Z])', r'\1_\2', s1).lower()

def geninstance(filename, timesteps):
    with open(filename) as f:
        with open("instance.lp", 'w') as g:
            section = "none"
            maxpc = 0
            maxtime = timesteps
            for line in f:
                line = line.strip()
                if line == ".code":
                    section = "code"
                    continue
                elif line == ".data":
                    section = "data"
                    continue
                elif section == "code":
                    ins = line.split(" ")
                    if ins[1] == "APICALL":
                        if ins[2].lower() == "exit":
                            g.write("end({:d}).\n".format(int(ins[0], 0)))
                        else:
                            g.write("apicall({:d}, {:s}).\n".format(int(ins[0], 0), convert_
                    elif ins[1] == "MOV":
                        try:
                            imm = int(ins[2], 0)
                            if imm == 0:
                                g.write("mov_imm({:d}, {:s}, zero).\n".format(int(ins[0], 0)
                            else:
                                for i in range(0, imm.bit_length()):
                                    if imm & (1<<i) != 0:
                                        g.write("mov_imm({:d}, {:s}, {:d}).\n".format(int(i
                        except ValueError:
```

```
        g.write("mov({:d}, {:s}, {:s}).\n".format(int(ins[0], 0), ins[2]
elif ins[1] == "CMP":
    try:
        imm = int(ins[3], 0)
        if imm == 0:
            g.write("cmp_imm({:d}, {:s}, zero).\n".format(int(ins[0], 0)
        else:
            for i in range(0, imm.bit_length()):
                if imm & (1<<i) != 0:
                    g.write("cmp_imm({:d}, {:s}, {:d}).\n".format(int(in
    except ValueError:
        g.write("cmp({:d}, {:s}, {:s}, cmp).\n".format(int(ins[0], 0), i
elif ins[1] == "ADD" or ins[1] == "SUB":
    try:
        imm = int(ins[3], 0)
        if imm == 0:
            g.write("{:s}_imm({:d}, {:s}, zero, {:s}).\n".format(ins[1].
        else:
            for i in range(0, imm.bit_length()):
                if imm & (1<<i) != 0:
                    g.write("{:s}_imm({:d}, {:s}, {:d}, {:s}).\n".format
    except ValueError:
        g.write("{:s}({:d}, {:s}, {:s}, {:s}).\n".format(ins[1].lower(),
elif ins[1][0] == 'J':
    try:
        imm = int(ins[2], 0)
        if imm == 0:
            g.write("{:s}_abs({:d}, zero).\n".format(ins[1].lower(), int
        else:
            for i in range(0, imm.bit_length()):
                if imm & (1<<i) != 0:
                    g.write("{:s}_abs({:d}, {:d}).\n".format(ins[1].lowe
    except ValueError:
        g.write("{:s}_reg({:d}, {:s}).\n".format(ins[1].lower(), int(ins
elif ins[1] == 'STO':
    try:
        imm = int(ins[3], 0)
        if imm == 0:
            g.write("sto_imm({:d}, {:s}, zero).\n".format(int(ins[0], 0)
        else:
            for i in range(0, imm.bit_length()):
                if imm & (1<<i) != 0:
                    g.write("sto_imm({:d}, {:s}, {:d}).\n".format(int(in
    except ValueError:
        g.write("sto({:d}, {:s}, {:s}).\n".format(int(ins[0], 0), ins[2]
elif ins[1] == 'EICAR':
```

```
                    g.write("{:s}({:d}).\n".format(ins[1].lower(), int(ins[0], 0)))
                maxpc = max(maxpc, int(ins[0], 0))
            if timesteps == 0:
                maxtime = 2*maxpc
            g.write("\n#const maxpc = {:d}.\n#const maxtime = {:d}.\n".format(maxpc, maxtime
            return maxpc
```

```
Overwriting geninstance.py
```

Computing PC¶

In order to understand these numeric PC values, an lp file must also be generated to convert the bitwise representation of the EIP to these numbers.

In [3]:

```
%%file genpcat.py

def genpcat(bits, maxpc):
    with open('pcat' + str(bits) + '.lp', 'w') as f:
        f.write('toolong(T, Reg) :-\n')
        f.write('    time(T),\n')
        f.write('    register(Reg),\n')
        f.write('    regbit(T, Reg, Pos),\n')
        f.write('    pos(Pos),\n')
        f.write('    Pos >= ' + str(maxpc.bit_length()) + '.\n\n')
        for i in range(0, maxpc + 1):
            f.write('pcat(T, ' + str(i) + ') :-\n')
            f.write('    time(T),\n')
            f.write('    pc(PC),\n')
            for j in range(0, maxpc.bit_length()):
                f.write('    ' + ('' if i&(1<<j) else 'not ') + 'regbit(T, eip, ' + str(j) +
            f.write('    not toolong(T, eip).\n')
```

```
Overwriting genpcat.py
```

Putting these two programs together, we can generate the necessary files to reason about a problem instance.

In [4]:

```
%%file prep-instance.py

import sys, geninstance, genpcat

maxpc = geninstance.geninstance(sys.argv[1], int(sys.argv[2]) if len(sys.argv) > 2 and sys.a
genpcat.genpcat(32, maxpc)
```

```
Overwriting prep-instance.py
```

In [5]:

```
%run prep-instance.py instance-6.ins
```

In [6]:

```
!cat instance.lp
```

```
apicall(1, get_something).
mov(2, eax, ebx).
apicall(3, get_something).
add(4, eax, ebx, eax).
cmp_imm(5, eax, 0).
cmp_imm(5, eax, 2).
cmp_imm(5, eax, 4).
cmp_imm(5, eax, 7).
cmp_imm(5, eax, 12).
cmp_imm(5, eax, 15).
cmp_imm(5, eax, 16).
cmp_imm(5, eax, 22).
jl_abs(6, 0).
jl_abs(6, 1).
jl_abs(6, 3).
cmp_imm(8, ebx, 3).
cmp_imm(8, ebx, 7).
jge_abs(9, 0).
eicar(10).
end(11).

#const maxpc = 11.
#const maxtime = 22.
```

In [7]:

```
!head pcat32.lp
```

```
toolong(T, Reg) :-
    time(T),
    register(Reg),
    regbit(T, Reg, Pos),
    pos(Pos),
    Pos >= 4.

pcat(T, 0) :-
    time(T),
    pc(PC),
```

Main Formulation¶

Now, the x86 processor and instruction set must be modeled. The lp file below describes a sufficient subset of the x86 processor and instruction set for the above instance.

In [8]:

```
%%file formulation.lp

#const intlen = 32.

register(eax;ebx;ecx;edx;esi;edi;edp;esp;eip;imm;cmp).
flag(c;z;s;o).
apitype_value(get_something).
apitype_alloc(alloc_something).
pos(0..intlen-1).
time(0..maxtime).
pc(0..maxpc).

%% Generalizations
mathop(T, cmp) :- time(T), pcat(T, PC), cmp_imm(PC, _, _).
mathop(T, Reg) :- time(T), pcat(T, PC), add(PC, _, _, Reg).
mathop(T, Reg) :- time(T), pcat(T, PC), sub(PC, _, _, Reg).
imm(T) :- time(T), pcat(T, PC), cmp_imm(PC, _, _).

%% Jump detection
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jmp_abs(PC, Pos).
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jo_abs(PC, Pos), flg(T, o).
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jno_abs(PC, Pos), not flg(T, o).
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), js_abs(PC, Pos), flg(T, s).
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jns_abs(PC, Pos), not flg(T, s).
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jz_abs(PC, Pos), flg(T, z).
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jnz_abs(PC, Pos), not flg(T, z).
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jc_abs(PC, Pos), flg(T, c).
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jnc_abs(PC, Pos), not flg(T, c).
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jl_abs(PC, Pos), flg(T, s), not flg(T, 
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jl_abs(PC, Pos), not flg(T, s), flg(T, 
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jge_abs(PC, Pos), flg(T, s), flg(T, o).
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jge_abs(PC, Pos), not flg(T, s), not flg
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jle_abs(PC, Pos), flg(T, s), not flg(T,
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jle_abs(PC, Pos), not flg(T, s), flg(T,
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jle_abs(PC, Pos), flg(T, z).
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jle_abs(PC, Pos), not flg(T, s), not flg
jump_abs(T, Pos) :- time(T), pos(Pos), pcat(T, PC), jle_abs(PC, Pos), flg(T, s), flg(T, o),
%jump(T) :- time(T), pcat(T, PC), jmp_rel(PC, _).
```

```
%jump(T) :- time(T), pcat(T, PC), jo_rel(PC, _).
%jump(T) :- time(T), pcat(T, PC), jno_rel(PC, _).
%jump(T) :- time(T), pcat(T, PC), js_rel(PC, _).
%jump(T) :- time(T), pcat(T, PC), jns_rel(PC, _).
%jump(T) :- time(T), pcat(T, PC), jz_rel(PC, _).
%jump(T) :- time(T), pcat(T, PC), jnz_rel(PC, _).
%jump(T) :- time(T), pcat(T, PC), jc_rel(PC, _).
%jump(T) :- time(T), pcat(T, PC), jnc_rel(PC, _).
%jump(T) :- time(T), pcat(T, PC), jl_rel(PC, _).
%jump(T) :- time(T), pcat(T, PC), jle_rel(PC, _).
%jump(T) :- time(T), pcat(T, PC), jg_rel(PC, _).
%jump(T) :- time(T), pcat(T, PC), jge_rel(PC, _).
%jump(T) :- time(T), pcat(T, PC), jmp_reg(PC, _).
%jump(T) :- time(T), pcat(T, PC), jo_reg(PC, _).
%jump(T) :- time(T), pcat(T, PC), jno_reg(PC, _).
%jump(T) :- time(T), pcat(T, PC), js_reg(PC, _).
%jump(T) :- time(T), pcat(T, PC), jns_reg(PC, _).
%jump(T) :- time(T), pcat(T, PC), jz_reg(PC, _).
%jump(T) :- time(T), pcat(T, PC), jnz_reg(PC, _).
%jump(T) :- time(T), pcat(T, PC), jc_reg(PC, _).
%jump(T) :- time(T), pcat(T, PC), jnc_reg(PC, _).
%jump(T) :- time(T), pcat(T, PC), jl_reg(PC, _).
%jump(T) :- time(T), pcat(T, PC), jle_reg(PC, _).
%jump(T) :- time(T), pcat(T, PC), jg_reg(PC, _).
%jump(T) :- time(T), pcat(T, PC), jge_reg(PC, _).

%% Registers maintain value unless acted upon
regbit(T, Reg, Pos) :-
    time(T),
    register(Reg),
    pos(Pos),
    regbit(T-1, Reg, Pos),
    pcat(T, PC),
    not end(PC),
    not sub(PC, _, _, Reg),
    not add(PC, _, _, Reg),
    not mov(PC, _, Reg),
    Reg != eip,
    Reg != eax,
    Reg != imm,
    Reg != cmp.
regbit(T, eax, Pos) :-
    time(T),
    pos(Pos),
    regbit(T-1, eax, Pos),
    pcat(T, PC),
    not end(PC),
    not apicall(PC, _),
    not sub(PC, _, _, eax),
    not add(PC, _, _, eax),
    not mov(PC, _, eax).

%% PC advances each time step
regbit(T+1, eip, 0) :-
    time(T),
    not regbit(T, eip, 0),
    not jump_abs(T, _).
regbit(T+1, eip, Pos) :-
    time(T),
    pos(Pos),
    Pos > 0,
    regbit(T, eip, Pos),
    regbit(T+1, eip, Pos-1),
    not jump_abs(T, _).
regbit(T+1, eip, Pos) :-
    time(T),
    pos(Pos),
    Pos > 0,
    regbit(T, eip, Pos),
    not regbit(T, eip, Pos-1),
    not jump_abs(T, _).
regbit(T+1, eip, Pos) :-
    time(T),
    pos(Pos),
    Pos > 0,
    not regbit(T, eip, Pos),
    not regbit(T+1, eip, Pos-1),
    regbit(T, eip, Pos-1),
    not jump_abs(T, _).

% Set PC on Jump
regbit(T+1, eip, Pos) :-
    time(T),
    pos(Pos),
    jump_abs(T, Pos).

% Comparisons

%% Store value from cmp_imm in imm
regbit(T, imm, Pos) :-
    time(T),
    pos(Pos),
    pcat(T, PC),
    cmp_imm(PC, _, Pos).

%% Reg - imm -> cmp
cmp(PC, Reg, imm, cmp) :- cmp_imm(PC, Reg, _).

%% MOV
regbit(T+1, Res, Pos) :-
    time(T),
    register(Res),
    pos(Pos),
    pcat(T, PC),
    mov(PC, Reg, Res),
    register(Reg),
    regbit(T, Reg, Pos).

%% Subtraction
regbit(T, Res, Pos) :-
    time(T),
    register(Res),
    pos(Pos),
    pcat(T, PC),
    cmp(PC, Reg1, Reg2, Res),
    register(Reg1),
    register(Reg2),
    regbit(T, Reg1, Pos),
    not regbit(T, Reg2, Pos),
    not carry(T, Pos).
regbit(T, Res, Pos) :-
    time(T),
    register(Res),
    pos(Pos),
    pcat(T, PC),
    cmp(PC, Reg1, Reg2, Res),
    register(Reg1),
    register(Reg2),
    not regbit(T, Reg1, Pos),
    regbit(T, Reg2, Pos),
    not carry(T, Pos).
regbit(T, Res, Pos) :-
    time(T),
    register(Res),
    pos(Pos),
    pcat(T, PC),
    cmp(PC, Reg1, Reg2, Res),
    register(Reg1),
    register(Reg2),
    regbit(T, Reg1, Pos),
    regbit(T, Reg2, Pos),
    carry(T, Pos).
regbit(T, Res, Pos) :-
    time(T),
    register(Res),
    pos(Pos),
    pcat(T, PC),
    cmp(PC, Reg1, Reg2, Res),
    register(Reg1),
    register(Reg2),
    not regbit(T, Reg1, Pos),
    not regbit(T, Reg2, Pos),
    carry(T, Pos).
carry(T, Pos+1) :-
    time(T),
    pos(Pos),
    pcat(T, PC),
    cmp(PC, Reg1, Reg2, _),
    register(Reg1),
    register(Reg2),
    not regbit(T, Reg1, Pos),
    regbit(T, Reg2, Pos).
carry(T, Pos+1) :-
    time(T),
    pos(Pos),
    pcat(T, PC),
    cmp(PC, Reg1, Reg2, _),
    register(Reg1),
    register(Reg2),
    regbit(T, Reg1, Pos),
    regbit(T, Reg2, Pos),
    carry(T, Pos).
carry(T, Pos+1) :-
    time(T),
    pos(Pos),
    pcat(T, PC),
    cmp(PC, Reg1, Reg2, _),
    register(Reg1),
    register(Reg2),
    not regbit(T, Reg1, Pos),
    not regbit(T, Reg2, Pos),
    carry(T, Pos).
regbit(T+1, Res, Pos) :-
    time(T),
    register(Res),
    pos(Pos),
    pcat(T, PC),
    sub(PC, Reg1, Reg2, Res),
    register(Reg1),
    register(Reg2),
    regbit(T, Reg1, Pos),
    not regbit(T, Reg2, Pos),
    not carry(T, Pos).
regbit(T+1, Res, Pos) :-
    time(T),
    register(Res),
    pos(Pos),
    pcat(T, PC),
    sub(PC, Reg1, Reg2, Res),
    register(Reg1),
    register(Reg2),
    not regbit(T, Reg1, Pos),
    regbit(T, Reg2, Pos),
    not carry(T, Pos).
regbit(T+1, Res, Pos) :-
    time(T),
    register(Res),
    pos(Pos),
    pcat(T, PC),
    sub(PC, Reg1, Reg2, Res),
    register(Reg1),
    register(Reg2),
    regbit(T, Reg1, Pos),
    regbit(T, Reg2, Pos),
    carry(T, Pos).
regbit(T+1, Res, Pos) :-
    time(T),
    register(Res),
    pos(Pos),
    pcat(T, PC),
    sub(PC, Reg1, Reg2, Res),
    register(Reg1),
    register(Reg2),
    not regbit(T, Reg1, Pos),
    not regbit(T, Reg2, Pos),
    carry(T, Pos).
carry(T, Pos+1) :-
    time(T),
    pos(Pos),
    pcat(T, PC),
    sub(PC, Reg1, Reg2, _),
    register(Reg1),
    register(Reg2),
    not regbit(T, Reg1, Pos),
    regbit(T, Reg2, Pos).
carry(T, Pos+1) :-
    time(T),
    pos(Pos),
    pcat(T, PC),
    sub(PC, Reg1, Reg2, _),
    register(Reg1),
    register(Reg2),
    regbit(T, Reg1, Pos),
    regbit(T, Reg2, Pos),
    carry(T, Pos).
carry(T, Pos+1) :-
    time(T),
    pos(Pos),
    pcat(T, PC),
    sub(PC, Reg1, Reg2, _),
    register(Reg1),
    register(Reg2),
    not regbit(T, Reg1, Pos),
    not regbit(T, Reg2, Pos),
    carry(T, Pos).

%% Addition
regbit(T+1, Res, Pos) :-
    time(T),
    register(Res),
    pos(Pos),
    pcat(T, PC),
    add(PC, Reg1, Reg2, Res),
    register(Reg1),
    register(Reg2),
    regbit(T, Reg1, Pos),
    not regbit(T, Reg2, Pos),
    not carry(T, Pos).
regbit(T+1, Res, Pos) :-
    time(T),
    register(Res),
    pos(Pos),
    pcat(T, PC),
    add(PC, Reg1, Reg2, Res),
    register(Reg1),
    register(Reg2),
    not regbit(T, Reg1, Pos),
    regbit(T, Reg2, Pos),
    not carry(T, Pos).
regbit(T+1, Res, Pos) :-
    time(T),
    register(Res),
    pos(Pos),
    pcat(T, PC),
    add(PC, Reg1, Reg2, Res),
    register(Reg1),
    register(Reg2),
    regbit(T, Reg1, Pos),
    regbit(T, Reg2, Pos),
    carry(T, Pos).
regbit(T+1, Res, Pos) :-
    time(T),
    register(Res),
    pos(Pos),
    pcat(T, PC),
    add(PC, Reg1, Reg2, Res),
    register(Reg1),
    register(Reg2),
    not regbit(T, Reg1, Pos),
    not regbit(T, Reg2, Pos),
    carry(T, Pos).
carry(T, Pos+1) :-
    time(T),
    pos(Pos),
    pcat(T, PC),
    add(PC, Reg1, Reg2, _),
    register(Reg1),
    register(Reg2),
    regbit(T, Reg1, Pos),
    regbit(T, Reg2, Pos).
carry(T, Pos+1) :-
    time(T),
    pos(Pos),
    pcat(T, PC),
    add(PC, Reg1, Reg2, _),
    register(Reg1),
    register(Reg2),
    regbit(T, Reg1, Pos),
    carry(T, Pos).
carry(T, Pos+1) :-
    time(T),
    pos(Pos),
    pcat(T, PC),
    add(PC, Reg1, Reg2, _),
    register(Reg1),
    register(Reg2),
    regbit(T, Reg2, Pos),
    carry(T, Pos).

%% API Calls: single value
0 { regbit(T, eax, Pos) } 1 :-
    time(T),
    pos(Pos),
    pcat(T, PC),
    apicall(PC, Call),
    apitype_value(Call).

%% Flags
flg(T, c) :-
    time(T),
    mathop(T, _),
    carry(T, intlen).
flg(T, c) :-
    time(T),
    not mathop(T, _),
    flg(T-1, c).
flg(T, z) :-
    time(T),
    mathop(T, Reg),
    not regbit(T, Reg, _).
flg(T, z) :-
    time(T),
    not mathop(T, _),
    flg(T-1, z).
flg(T, s) :-
    time(T),
    mathop(T, Reg),
    regbit(T, Reg, intlen-1).
flg(T, s) :-
    time(T),
    not mathop(T, _),
    flg(T-1, s).
flg(T, o) :-
    time(T),
    pcat(T, PC),
    add(PC, Reg1, Reg2, Res),
    register(Reg1),
    register(Reg2),
    register(Res),
    regbit(T-1, Reg1, intlen-1),
    regbit(T-1, Reg2, intlen-1),
    not regbit(T, Res, intlen-1).
flg(T, o) :-
    time(T),
    pcat(T, PC),
    add(PC, Reg1, Reg2, Res),
    register(Reg1),
    register(Reg2),
    register(Res),
    not regbit(T-1, Reg1, intlen-1),
    not regbit(T-1, Reg2, intlen-1),
    regbit(T, Res, intlen-1).
flg(T, o) :-
    time(T),
    pcat(T, PC),
    sub(PC, Reg1, Reg2, Res),
    register(Reg1),
    register(Reg2),
    register(Res),
    regbit(T-1, Reg1, intlen-1),
    not regbit(T-1, Reg2, intlen-1),
    not regbit(T, Res, intlen-1).
flg(T, o) :-
    time(T),
    pcat(T, PC),
    sub(PC, Reg1, Reg2, Res),
    register(Reg1),
    register(Reg2),
    register(Res),
    not regbit(T-1, Reg1, intlen-1),
    regbit(T-1, Reg2, intlen-1),
    regbit(T, Res, intlen-1).
flg(T, o) :-
    time(T),
    not mathop(T, _),
    flg(T-1, o).
```

```
Overwriting formulation.lp
```

Malware detection¶

Now that we have a model for the processor, we must define what actions constitute malware. Then, we can forbid solutions that don't encounter any malware to determine if there is a way the software can be malicious.

In [9]:

```
%%file find-malware.lp

malware_sig_eicar(T, PC) :- time(T), pcat(T, PC), eicar(PC).
malware_sig_self_edit(T, PC) :- time(T), pcat(T, PC), sto(PC, Reg1, Reg2), register(Reg1), r

malware_sig(T) :- malware_sig_eicar(T, _).
malware_sig(T) :- malware_sig_self_edit(T, _).

:- not malware_sig(_).
%:- end(PC), #sum { T2:pcat(T2, PC); -T1:malware_sig(T1) } -1.
%#maximize { 1, sig, T: malware_sig(T) }.
```

```
Overwriting find-malware.lp
```

Retrieving results¶

Once a malware signature has been found, we want to identify the conditions for reaching it. To do so, we want to generate new facts that identify the results of API calls, and show them.

In [10]:

```
%%file diagnostics.lp

apicall_ret(T, PC) :-
    time(T),
    pcat(T, PC),
    apicall(PC, _).

apicall_res(T, Pos) :-
    apicall_ret(T, _),
    pos(Pos),
    regbit(T, eax, Pos).

#show apicall_ret/2.
#show apicall_res/2.
#show malware_sig_eicar/2.
#show malware_sig_self_edit/2.
```

```
Overwriting diagnostics.lp
```

Solving¶

Putting this all together, we can now solve.

In [11]:

```
!clingo formulation.lp instance.lp pcat32.lp find-malware.lp diagnostics.lp -W none -c intle
```

```
{
  "Solver": "clingo version 5.1.0",
  "Input": [
    "formulation.lp","instance.lp","pcat32.lp","find-malware.lp","diagnostics.lp"
  ],
  "Call": [
    {
      "Witnesses": [
        {
          "Value": [
            "apicall_ret(1,1)", "apicall_ret(3,3)", "apicall_res(3,31)", "apicall_res(1,31)'
          ]
        }
      ]
    }
  ],
  "Result": "SATISFIABLE",
  "Models": {
    "Number": 1,
    "More": "yes"
  },
  "Calls": 1,
  "Time": {
    "Total": 0.954,
    "Solve": 0.014,
    "Model": 0.014,
    "Unsat": 0.000,
    "CPU": 0.766
  }
}
```

Visualization¶

The result can now be visualized.

In [12]:

```
%%file visualizer.py
```

```
import json

maxtime = 16
maxpc = 8
maxtimelen = str(int(maxtime.bit_length()/4)+1)
maxpclen = str(int(maxpc.bit_length()/4)+1)

calls = {}
sigs = []
cpu = 0
grounding = 0
solving = 0

with open('trace.json') as f:
    data = json.loads(f.read())
    if "Witnesses" in data["Call"][0]:
        for fact in data["Call"][0]["Witnesses"][-1]["Value"]:
            ent = fact.split('(')
            ent[1] = ent[1][:-1].split(',')
            if ent[0] == "apicall_ret":
                if not int(ent[1][0]) in calls:
                    calls[int(ent[1][0])] = {'PC' : int(ent[1][1]), 'value': 0}
                else:
                    calls[int(ent[1][0])]['PC'] = int(ent[1][1])
            if ent[0] == "apicall_res":
                if not int(ent[1][0]) in calls:
                    calls[int(ent[1][0])] = {'PC' : 0, 'value': 1<<int(ent[1][1])}
                else:
                    calls[int(ent[1][0])]['value'] += 1<<int(ent[1][1])
            if ent[0][:12] == "malware_sig_":
                sigs.append({'type':ent[0][12:], 'time': int(ent[1][0]), 'PC': int(ent[1][1
    else:
        print("No malware signatures could be reached.")
    cpu = data["Time"]["CPU"]
    solving = data["Time"]["Solve"]
    grounding = cpu - solving

for sig in sigs:
    print(("Signature {:s} encountered at time 0x{:0" + maxtimelen + "X} at PC 0x{:0" + maxp
          .format(sig['type'].upper(), sig['time'], sig['PC']))
    print(("{:" + str(int(maxtimelen)+2) + "s}\t{:" + str(int(maxpclen)+2) + "s}\t{:10s}").f
    for call in sorted(calls.items()):
        print(("0x{:0" + maxtimelen + "X}\t0x{:0" + maxpclen + "X}\t0x{:08X}")\
             .format(call[0], call[1]['PC'], call[1]['value']))

print('\nTotal Time: {:1.03f}s\tGrounding: {:1.03f}s\tSolving: {:1.03f}s'.format(cpu, ground
Overwriting visualizer.py
```

In [13]:

```
%run visualizer.py
```

```
Signature EICAR encountered at time 0x0A at PC 0x0A

Time    PC      Value
0x01    0x01    0x80100000
0x03    0x03    0x80000000

Total Time: 0.766s  Grounding: 0.752s   Solving: 0.014s
```

Packaging¶

For ease of use, all of these commands are packaged into a bash script.

In [14]:

```
%%file aspemu
#!/bin/bash

filename="$1"
timesteps="$2"

if [ "$filename" = "" ]; then
    echo "usage: aspemu <filename> [timesteps]"
fi
python prep-instance.py "$filename" "$timesteps"
if [ "$?" -ne "0" ]; then
    exit
fi
clingo formulation.lp instance.lp pcat32.lp find-malware.lp diagnostics.lp -W none --outf=2
python visualizer.py
```

```
Overwriting aspemu
```

In [15]:

```
!bash aspemu instance-6.ins
```

```
Signature EICAR encountered at time 0x0A at PC 0x0A

Time    PC      Value
0x01    0x01    0x80100000
0x03    0x03    0x80000000

Total Time: 0.391s  Grounding: 0.375s   Solving: 0.016s
```

Performance Analysis

Performance Analysis¶

A Simple Formulation:¶

In [1]:

```
%%file instance-1.ins
.code
0x01 APICALL GetSomething
0x02 CMP EAX 0xDEADBEEF
0x03 JNZ 0x5
0x04 EICAR
0x05 APICALL Exit
```

```
Overwriting instance-1.ins
```

In [2]:

```
!bash aspemu instance-1.ins
```

```
Signature EICAR encountered at time 0x04 at PC 0x04

Time    PC      Value
0x01    0x01    0xDEADBEEF

Total Time: 0.109s  Grounding: 0.109s   Solving: 0.000s
```

All time is in grounding.

More Complex Formulations¶

In [3]:

```
%%file instance-2.ins
.code
0x01 APICALL GetSomething
0x02 CMP EAX 0xDEADBEEF
```

```
0x03 JNZ 0x8
0x04 APICALL GetSomething
0x05 CMP EAX 0xDEADBEEF
0x06 JNZ 0x8
0x07 EICAR
0x08 APICALL Exit
```

```
Overwriting instance-2.ins
```

In [4]:

```
!bash aspemu instance-2.ins
```

```
Signature EICAR encountered at time 0x07 at PC 0x07

Time    PC      Value
0x01    0x01    0xDEADBEEF
0x04    0x04    0xDEADBEEF

Total Time: 0.203s  Grounding: 0.200s   Solving: 0.003s
```

In [5]:

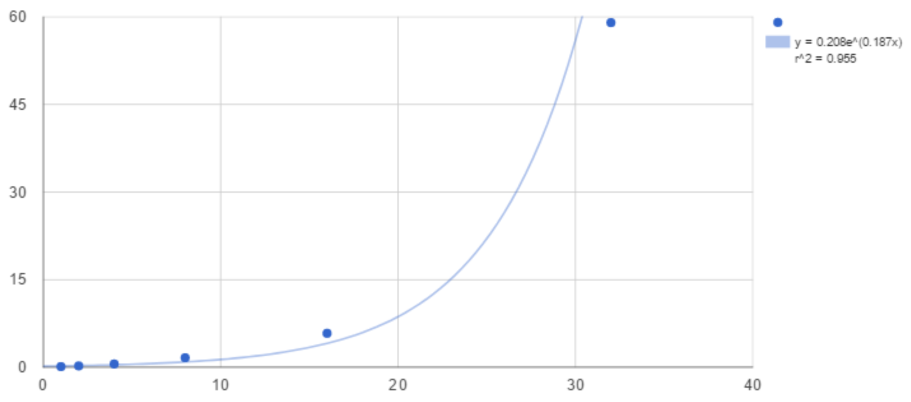
```
import subprocess

for i in range(6):
    with open('perftest.ins', 'w') as f:
        PC = 1
        f.write(".code\n")
        for j in range(1<<i):
            f.write("0x{:02X} APICALL GetSomething\n".format(PC))
            f.write("0x{:02X} CMP EAX 0xDEADBEEF\n".format(PC+1))
            f.write("0x{:02X} JNZ 0x{:X}\n".format(PC+2, (3 * (1<<i)) + 3))
            PC = PC + 3
        f.write("0x{:02X} EICAR\n".format(PC))
        f.write("0x{:02X} APICALL Exit\n".format((3 * (1<<i)) + 3))
    print(str(1<<i) + ": ", end='')
    print(str(subprocess.check_output("bash aspemu perftest.ins | grep Total", shell=True))
```

```
1: Total Time: 0.141s  Grounding: 0.140s   Solving: 0.001s
2: Total Time: 0.297s   Grounding: 0.292s   Solving: 0.005s
4: Total Time: 0.750s   Grounding: 0.616s   Solving: 0.134s
8: Total Time: 1.813s   Grounding: 1.688s   Solving: 0.125s
16: Total Time: 6.203s  Grounding: 5.501s   Solving: 0.702s
32: Total Time: 58.703s Grounding: 19.543s  Solving: 39.160s
```

In [6]:

```
from IPython.display import Image
Image("performance.png")
```

Out[6]:



Time scales exponentially with number of API calls, however the constant is small, so it is still reasonable to run.

Automated Tests

Automated Tests¶

In [1]:

```
%%file tests.yaml

Definitions:
    emu: {filename: formulation.lp}
    inst: {filename: instance.lp}
    both: {group: [emu, inst]}

Test Full Program On Instance:
    Modules: both
    Expect: SAT

Test PC Advance:
    Modules: emu
    inline: |
        #const maxpc = 2.
        #const maxtime = 2.
        end(2).
        advance :-
            pcat(1, 1),
            pcat(0, 0).
        :- not advance.
    Expect: SAT

Test Z Flag CMP:
    Modules: emu
    inline: |
        #const maxtime = 2.
        #const maxpc = 2
        cmp_imm(1, eax, zero).
        end(2).
        :- not flg(2, z).
    Expect: SAT

Test NZ Flag CMP:
    Modules: emu
    inline: |
        #const maxtime = 2.
        #const maxpc = 2
        cmp_imm(1, eax, 0).
        end(2).
        :- flg(2, z).
    Expect: SAT

Test S Flag CMP:
    Modules: emu
    inline: |
        #const maxtime = 2.
        #const maxpc = 2
        cmp_imm(1, eax, 0).
        end(2).
        :- not flg(2, s).
    Expect: SAT

Test NS Flag CMP:
    Modules: emu
    inline: |
        #const maxtime = 2.
        #const maxpc = 2
        cmp_imm(1, eax, 0..intlen-1).
        end(2).
        :- flg(2, s).
    Expect: SAT

Test C Flag CMP:
    Modules: emu
    inline: |
        #const maxtime = 2.
        #const maxpc = 2
        cmp_imm(1, eax, 0).
        end(2).
        :- not flg(2, c).
    Expect: SAT

Test NC Flag CMP:
    Modules: emu
    inline: |
        #const maxtime = 2.
        #const maxpc = 2
        cmp_imm(1, eax, 0).
        end(2).
        :- flg(2, c).
    Expect: SAT
```

```
Overwriting tests.yaml
```

In [2]:

```
!ansunit tests.yaml -vv
```

```
Test tests.yaml :: C Flag CMP ... ok
Test tests.yaml :: Full Program On Instance ... ok
Test tests.yaml :: NC Flag CMP ... ok
Test tests.yaml :: NS Flag CMP ... ok
Test tests.yaml :: NZ Flag CMP ... ok
Test tests.yaml :: PC Advance ... ok
Test tests.yaml :: S Flag CMP ... ok
Test tests.yaml :: Z Flag CMP ... ok

----------------------------------------------------------------------
Ran 8 tests in 1.008s

OK
```