PyEmu: A multi-purpose scriptable IA-32 emulator

Cody Pierce

TippingPoint DVLabs
cpierce@tippingpoint.com

1 Introduction

Emulators have existed since the modern computer systems they emulate. In 1965 IBM released the first computer system based entirely on integrated circuits[1]. With it they packaged an emulator to aid in its adoption. In modern days, emulators appear in all sorts of applications. These applications range from complete virtual machines to old arcade systems. In this paper, we will look at how the world of emulation pertains to, and helps the reverse engineering discipline.

When one looks at emulation in modern computer science, it can be broken down into what is perceived as two main methods of operation system emulation and instruction emulation.

1.1 System Emulation

System emulation is a very attractive method for doing complete replication of how a normal system operates. This includes not only emulating a processor and memory, but peripherals as well.

The most outstanding piece that differentiates this from instruction emulation is the peripheral emulation. Since the goal of system emulation is to provide a complete environment for core software, such as an operating systems, to be installed the emulator must handle requests to video cards, disk controllers, network devices, as well as providing a BIOS.

A good example of this type of emulation is the bochs[2] IA-32 emulator. It provides the user with the ability to install guest operating systems on a virtual disk managed by bochs. As stated previously, this type of full system emulation will act just like a physical computer, providing keyboard/mouse input and output, as well as other devices.

1.2 Instruction Emulation

The second form of emulation is what can be considered instruction emulation. In this sense instruction emulators only handle the tasks of translating CPU behavior to their equivalent logical and memory computations. This type of emulation is best suited for specific use and will be the focus of this paper.

Instruction emulation may seem limiting at first glance. However it is tailored to serve in the role of a tool, as opposed to a system emulator that works as an application. The benefit of this approach is openness and flexibility. While keeping the purpose basic, it allows the user to define what it is emulating with greater control.

2 Emulation as it applies to reverse engineering

Since the focus of this paper is emulation as applied to reverse engineering, one must look at the current state of affairs and applications of this technology. The state of reverse engineering is only getting more complex. While application continue to evolve and take on more features, the needed time to comprehend an application via reverse engineering greatly increases. These complexities often lead to frustration and hopelessness for someone trying to understand the assembly level actions of a program in static disassembly.

2.1 Complex code paths

An often insurmountable task when reversing software is complex code paths. Any given binary may contain thousands of difficult to understand and time consuming functions. Whether this appears as one large function, or hundreds of branches, the problem persists. Code path understanding is essential to the overall comprehension of a program's logic. Therefore, we may be able to utilize emulation to decipher cryptic nodes.

Take the following example of a complex code path as displayed in IDA[3]
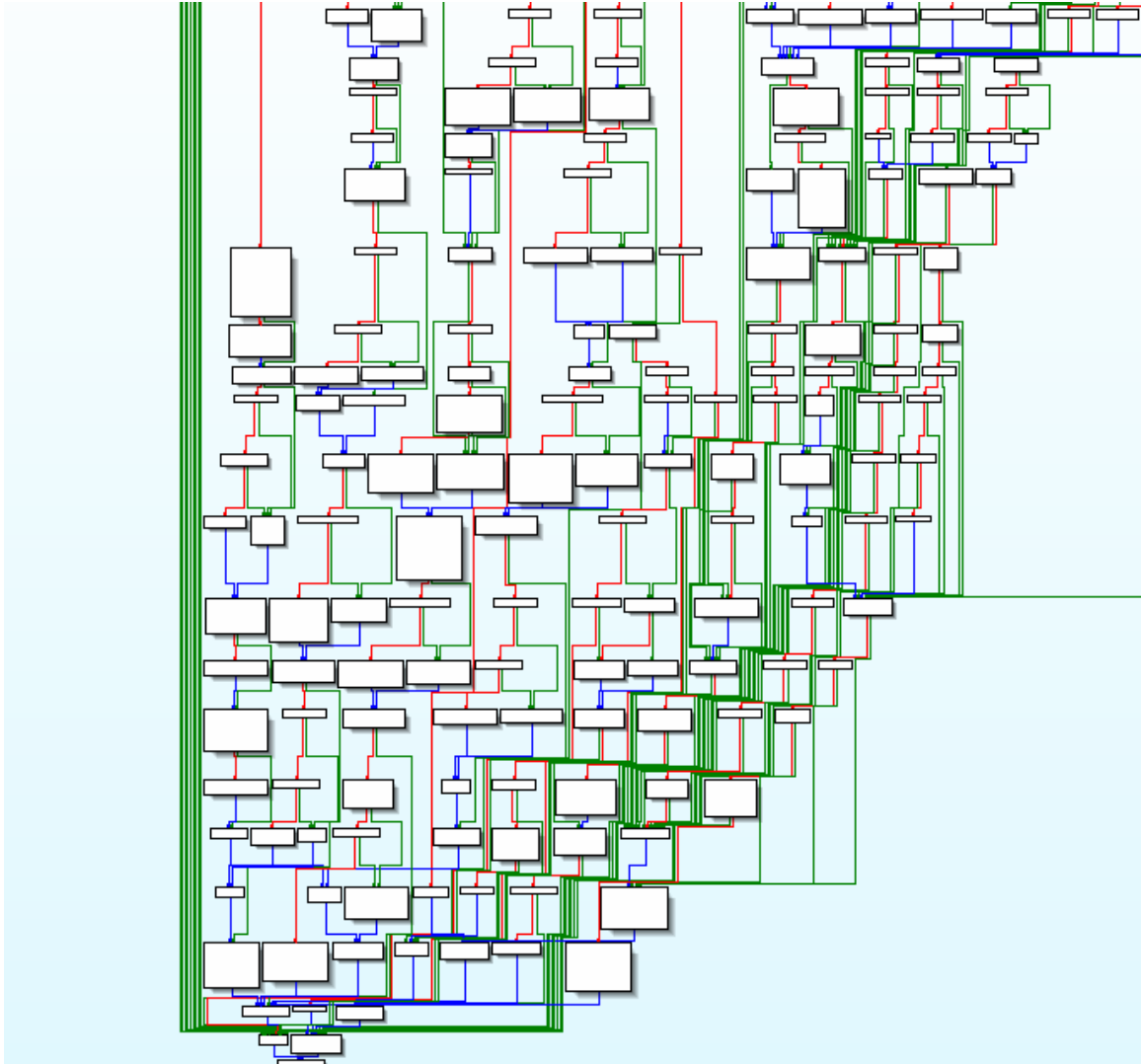
To statically reverse engineer this **single function** in the binary would take a large amount of time. Instead, the function arguments can be identified and the behavior of the code emulated. The results can then be used to determine the modifications and logic taken based upon this information. While this may seem like an oversimplification of a complex problem, it will be seen that PyEmu can easily achieve this through various methods.

2.2 Ambiguous Code

Another example we will briefly touch upon that hinders the process of reverse engineering is seemingly ambiguous code blocks. This is a very common side effect of doing static analysis on a program and is not usually a problem for live analysis with a debugger. However, in wanting to move to a purely static analysis method without having to fall back on live debugging, these problems must be addressed.

An ambiguous code block example

```
      loc_3056C46F:
018 push     ebx
01C call     sub_30025460
018 push     [ebp+11ECh+var_1174]
01C lea      eax, [ebp+11ECh+var_828]
01C push     esi
020 push     eax
024 call     sub_30566839
018 push     ebx
01C call     sub_30025460
018 push     dword ptr [edi+0Ch]
01C call     sub_30025460
018 lea      eax, [ebp+11ECh+var_828]
018 push     eax
01C call     sub_30565A22
018 push     ebx
01C call     sub_30025460
018 push     dword ptr [edi+0Ch]
01C call     sub_30025460
018 cmp      dword_3087B628, esi
018 push     2
01C pop      ebx
018 jnz      short loc_3056C4C6
```
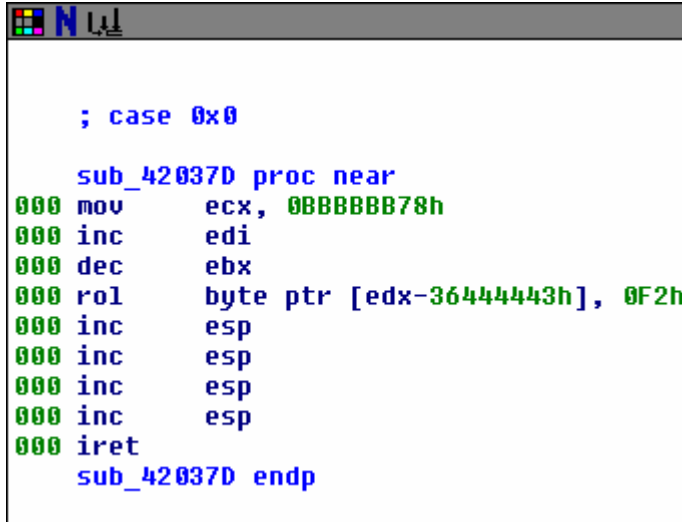
This code snippet of a basic block really does not mean much to the naked eye. Even with the rest of the function in tact, this block has 7 branches, various local variables, and what appears to be an object or structure of some kind. Currently, no tools exist to aid a researcher in organizing and understanding this basic block. In this case a scriptable emulator will help greatly, by making the reverse engineering process more efficient.

2.3 Code Obfuscation

While not necessarily common in production services, code obfuscation is gaining significant ground with companies trying to protect intellectual property. With the emergence and proliferation of reverse engineering as a means to gain an advantage over a competitor, often times a company may add road blocks to deter this and retain closely guarded secrets.

Code obfuscation techniques vary wildly from deceptive anti-disassembly methods tricking disassemblers and debuggers, to hand implemented functions to deceive a potential reverse engineer. As this becomes commonplace in software, one must have a means of quickly reducing the complexity reveal the meaning of such things.

A simple example of obfuscation

```
; case 0x0

    sub_42037D proc near
000 mov     ecx, 0BBBBBB78h
000 inc     edi
000 dec     ebx
000 rol     byte ptr [edx-36444443h], 0F2h
000 inc     esp
000 inc     esp
000 inc     esp
000 inc     esp
000 iret
    sub_42037D endp
```

The above example demonstrates a potential attempt to thwart any onlookers as to what really might be happening. It could also be an attempt to prevent the disassembler from properly analyzing the target binary. In this instance one can use an emulator to run all code paths leading to this function and observe any values modified during its run. This can potentially speed up the process of determining how the values are being used, or if they have any significance at all.

2.4 Time

Time is the single most valuable and exploitable resource related to reverse engineering. Advancing the field must always include reducing the time it takes to fully examine pieces of a binary and reach the mythical 100% code coverage goal. This will be achieved with a combination of scripts and tools helping focus manual analysis.

It is hard to quantify the time it may take to completely understand a given binary. Many factors must be considered when determining how much time may be spent. Size, complication, proficiency, and organize all play major roles in the time equation when reverse engineering. The following example is a snapshot of a major piece of software and its number of functions.

Functions window

Function name	Segment	Start	Length	R	F	L	S	B	T	=
sub_303F5508	.text	303F5508	00001467	R	.	.	.	B	.	.
sub_305690CE	.text	305690CE	0000147C	R	.	.	.	B	T	.
sub_3068AF6D	.text	3068AF6D	00001489	R	.	.	.	B	.	.
sub_3056BD48	.text	3056BD48	00001498	R	.	.	.	B	T	.
sub_30573FC6	.text	30573FC6	0000150B	R	.	.	.	B	.	.
sub_303AE2C4	.text	303AE2C4	0000153C	R	.	.	.	B	T	.
sub_3034C00E	.text	3034C00E	00001610	R	.	.	.	B	.	.
sub_3056A725	.text	3056A725	00001623	R	.	.	.	B	.	.
sub_30823F95	.text	30823F95	00001659	R	.	.	.	.	.	.
sub_30533567	.text	30533567	000016A9	R	.	.	.	B	.	.
sub_3080EA3A	.text	3080EA3A	000016E3	R	.	.	.	B	.	.
sub_3046F6D3	.text	3046F6D3	000016F8	R	.	.	.	B	.	.
sub_30653511	.text	30653511	000017CB	R	.	.	.	B	.	.
sub_303F0500	.text	303F0500	00001AF1	R	.	.	.	B	.	.
sub_3058284D	.text	3058284D	00001CF3	R	.	.	.	B	.	.
sub_305B4154	.text	305B4154	00001E0F	R	.	.	.	B	T	.
sub_3045EB4D	.text	3045EB4D	00001E63	R	.	.	.	B	T	.
sub_304306E5	.text	304306E5	000021B7	R	.	.	.	B	.	.
sub_30118D16	.text	30118D16	00002214	R	.	.	.	B	T	.
sub_3058CCE5	.text	3058CCE5	0000225E	R	.	.	.	B	T	.
sub_301F8A44	.text	301F8A44	0000288B	R	.	.	.	.	T	.
sub_304A9CAD	.text	304A9CAD	000028FE	R	.	.	.	B	.	.
sub_304D98F9	.text	304D98F9	00002BB9	R	.	.	.	B	.	.
sub_304D5098	.text	304D5098	00002BE7	R	.	.	.	B	.	.
sub_30635807	.text	30635807	00003281	R	.	.	.	B	.	.
sub_30845A57	.text	30845A57	00004A87	R	.	.	.	B	.	.

Line 27754 of 27754

As can be seen this binary has **27754** functions. Take note of the length of the sorted functions. In this example we see a functions of length 0x4A87 (19079) bytes! Assuming a skilled reverse engineer would take 10 minutes per function, an ambitious time frame, (this is ignoring the fact that **950** of the functions were well over 1024 bytes) the time taken to reverse this software is

$$((27754 * 10) / 60) / 24 = 193 \text{ days}$$

Assuming it would take 10 minutes per function is absurd, but even with superman at the helm reversing it would take him 193 straight days to completely understand 100% of this piece of software. As can plainly be seen, reducing the time to understand functions is a major priority. Emulation is one technique that can greatly help in this area.

2.5 Current Tools

The current list of available tools for reverse engineering, and Python based tools in particular grows daily. With professionally developed tools like BinNavi[4], open source community projects like PaiMei[5] and community contributed scripts and plugins for IDA Pro, there is a no shortage of options to help in the previous problems. However there currently exists only one emulator targeted at reverse engineering. The IDA Pro plugin idax86emu[6], by Chris Eagle, allows a user to add values to a stack, change and monitor registers, and even emulate library calls. While this is a very good plugin and its obvious benefits, it does lack flexibility and extensibility. The plugin being written in C,

as all IDA plugins are, can be a blessing or a curse. It is hardly debatable whether you can dynamically control, monitor, or modify values on the fly with the inherent quickness, and ease of a scripting language. It just does not allow one to easily expand and truly integrate it into their workflow.

3 PyEmu Architecture

Before going into architecture specifics related to PyEmu, we must first look at why Python as a programming language was chosen. Obviously, it is not common practice to emulate low level code in a high level language. Since low level assembly simply operates on basic computational logic I felt it would be straightforward to mimic this in a language such as Python. Also, another determining factor in the language choice was current progress in other Python tools.

Many people enjoy using Python and thus have created tools around it to aid in reverse engineering tasks. IDAPython[7] exists to allow script access to the IDA Pro scripting language (IDC) and plugin SDK. This alone allows for immeasurable amounts of options, one being the building of additional libraries on top of the language. One of these libraries is PIDA[8]. An abstraction library for quickly accessing structural information about the current binary disassembled in IDA.

Besides IDA, other tools exist when doing live analysis, and binary processing. Pydbg[9] is a python library that wraps the native win32 debugging API allowing a researcher to implement flexible scripts for controlling a debugee including execution, memory access, and context information such as registers. Pefile[10] is another library for processing PE executable file formats in Python. This library allows the parsing of important information pertaining to an executable for disassembling including imports, code, and data sections as well as entry point addresses. Finally, there is pydasm[11] which is a python interface to the disassembly library libdasm[12]. Pydasm can arbitrarily handle the disassembly of instructions and allow an emulator to be even more flexible in operation.

3.1 Overview

The PyEmu architecture works by providing the user with a flexible abstracted API in the form of the PyEmu class. This class will handle execution of instructions, fetching of memory, and any user requested information. The PyEmu architecture is separated into three classes including PyCPU, PyMemory, and of course PyEmu. By separating these aspects of an emulator, we can provide control over how each subsystem operates. This power is the essence of PyEmu. As a user, the ability to control memory allocations, instruction execution, and execution via clean methods is fantastically flexible.

When PyEmu is tasked with executing, it instructs PyCPU to execute a single instruction. PyCPU will request the memory for that instruction from PyEMU which will then pass the request to PyMemory. The reverse happens when the request is returned.

This may seem non-intuitive, but because the user is allowed to control all aspects of this process via PyEmu, it is required. The interaction is demonstrated below.
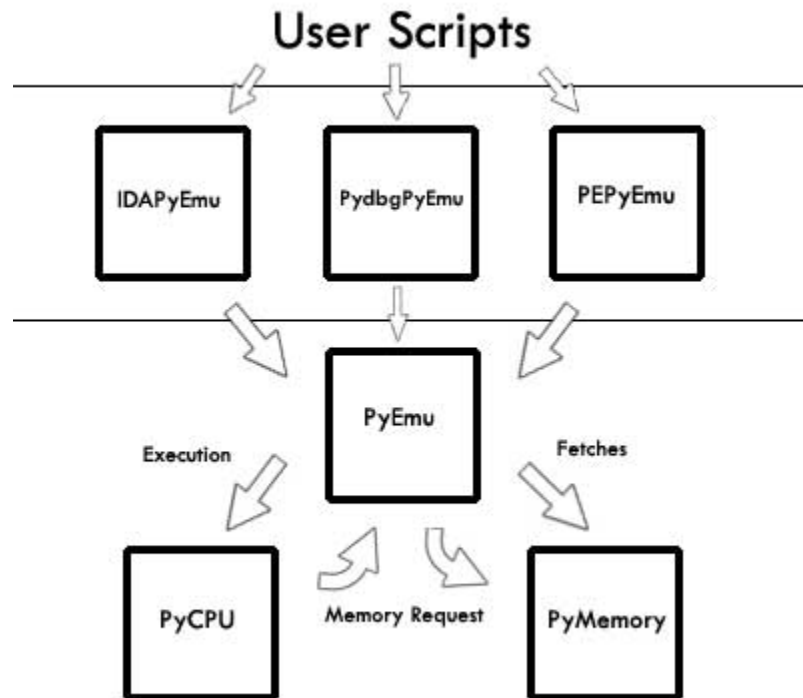
This is the core modus operandi of the PyEmu package. In general, a user should only interface with PyEmu classes, all useful information is exposed through public methods when instantiating the PyEmu derived class. Nothing is to say one may not want to create new classes and new layers of abstraction.

3.1 PyCPU

PyCPU is the heart of the PyEmu emulator, PyCPU handles all of the instruction logic, execution and related processor tasks. The job of the CPU is to execute a given instruction based strictly on the Intel reference specification[13]. As with every piece of the PyEmu architecture, the CPU code tries to autonomously handle all of these needed functions.

The most basic action is to execute an instruction. To fetch this instruction we access memory for the current instruction pointer address and hand it to pydasm. Pydasm allows us to properly decode the wanted instruction into its opcode, and operands. This simple operation is the essence of PyEmu. Allowing the emulator to arbitrarily decode instructions helps serve various function in any environment we want, including live analysis through Pydbg or statically via IDA Pro.

```python
    def execute(self):
        # Save our old instruction pointer
        oldeip = self.EIP

        # Fetch raw instruction from memory
        rawinstruction = self.get_memory(self.EIP, 32)
        if not rawinstruction:
            print "[!] Problem fetching raw bytes from 0x%08x" %
(self.EIP)

            return False

        # Decode instruction from raw returning a pydasm.instruction
        instruction = pydasm.get_instruction(rawinstruction,
pydasm.MODE_32)
        if not instruction:
            print "[!] Problem decoding instruction"

            return False

        pyinstruction = PyInstruction(instruction)
```

As can be seen once we have grabbed the next instruction we can then call into our proper mnemonic handler.

```python
        pyinstruction.mnemonic = pyinstruction.mnemonic.split()[0]

        if pyinstruction.mnemonic in self.supported_instructions:
            if not
self.supported_instructions[pyinstruction.mnemonic](pyinstruction):
                return False
        else:
            print "[!] Unsupported instruction %s" %
pyinstruction.mnemonic
            return False

        if self.EIP == oldeip:
            #print "[*] Updating eip from 0x%08x -> 0x%08x" %
(self.EIP, self.EIP + pyinstruction.length)
            self.EIP += pyinstruction.length

        return True
```

PyEmu uses mnemonics so that it can cleanly handle groups of opcodes within a mnemonic. Since mnemonics have multiple ways of operating this allows developers extending PyCPU to easily add more if necessary and implement mnemonic and opcode level hooking. Currently PyEmu supports 100+ Intel IA-32 instructions. While one might note that the Intel specification contains over 400 mnemonics, its important to understand that in most cases roughly 50 instructions are ever used in real world applications.

Once PyCPU has the proper mnemonic, it calls into the function that handles that particular instruction. PyEmu strives to be easy to extend, most of the mnemonic and opcode handling has been pre-generated allowing for a uniform look and operation of all instructions.

```python
def CMP(self, instruction):
    op1 = instruction.op1
    op2 = instruction.op2
    op3 = instruction.op3

    so = instruction.operand_so()
    ao = instruction.address_so()

    op1value = ""
    op2value = ""
    op3value = ""
    op1valuederef = None
    op2valuederef = None

    #38 /r CMP r/m8,r8 Compare r8 with r/m8
    if instruction.opcode == 0x38:

        size = 1

        if op1.type == pydasm.OPERAND_TYPE_REGISTER:
            op1value = self.get_register(op1.reg, size)
            op2value = self.get_register(op2.reg, size)

            # Do logic
            result = op1value - op2value

            self.set_arithmetic_flags(op1value, op2value,
result, size)

        elif op1.type == pydasm.OPERAND_TYPE_MEMORY:
            op1value = self.get_memory_address(instruction,
1, size)
            op2value = self.get_register(op2.reg, size)

            # Do logic
            op1valuederef = self.get_memory(op1value, size)

            result = op1valuederef - op2value

            self.set_arithmetic_flags(op1valuederef,
op2value, result, size)

    #39 /r CMP r/m16,r16 Compare r16 with r/m16
    #39 /r CMP r/m32,r32 Compare r32 with r/m32
    elif instruction.opcode == 0x39:

        if so:
            size = 2
        else:
            size = 4
```

```python
            if op1.type == pydasm.OPERAND_TYPE_REGISTER:
                op1value = self.get_register(op1.reg, size)
                op2value = self.get_register(op2.reg, size)

                # Do logic
                result = op1value - op2value

                self.set_arithmetic_flags(op1value, op2value,
result, size)

            elif op1.type == pydasm.OPERAND_TYPE_MEMORY:
                op1value = self.get_memory_address(instruction,
1, size)

                op2value = self.get_register(op2.reg, size)

                # Do logic
                op1valuederef = self.get_memory(op1value, size)

                result = op1valuederef - op2value

                self.set_arithmetic_flags(op1valuederef,
op2value, result, size)

        #3B /r CMP r16,r/m16 Compare r/m16 with r16
        #3B /r CMP r32,r/m32 Compare r/m32 with r32
        elif instruction.opcode == 0x3b:

            if so:
                size = 2
            else:
                size = 4

            op1value = self.get_register(op1.reg, size)

            if op2.type == pydasm.OPERAND_TYPE_REGISTER:
                op2value = self.get_register(op2.reg, size)

                # Do logic
                result = op1value - op2value

                self.set_arithmetic_flags(op1value, op2value,
result, size)

            elif op2.type == pydasm.OPERAND_TYPE_MEMORY:
                op2value = self.get_memory_address(instruction,
2, size)

                # Do logic
                op2valuederef = self.get_memory(op2value, size)

                result = op1value - op2valuederef

                self.set_arithmetic_flags(op1value,
op2valuederef, result, size)

        #3C ib CMP AL, imm8 Compare imm8 with AL
        elif instruction.opcode == 0x3c:
```

```python
            size = 1

            op1value = self.get_register(0, size)
            op2value = op2.immediate & self.get_mask(size)

            # Do logic
            result = op1value - op2value

            self.set_arithmetic_flags(op1value, op2value, result,
size)

        #3D id CMP EAX, imm32 Compare imm32 with EAX
        #3D iw CMP AX, imm16 Compare imm16 with AX
        elif instruction.opcode == 0x3d:

            if so:
                size = 2
            else:
                size = 4

            op1value = self.get_register(0, size)
            op2value = op2.immediate & self.get_mask(size)

            # Do logic
            result = op1value - op2value

            self.set_arithmetic_flags(op1value, op2value, result,
size)

        #81 /7 id CMP r/m32,imm32 Compare imm32 with r/m32
        #81 /7 iw CMP r/m16, imm16 Compare imm16 with r/m16
        elif instruction.opcode == 0x81 and instruction.extindex
== 0x7:

            if so:
                size = 2
            else:
                size = 4

            if op1.type == pydasm.OPERAND_TYPE_REGISTER:
                op1value = self.get_register(op1.reg, size)
                op2value = op2.immediate & self.get_mask(size)

                # Do logic
                result = op1value - op2value

                self.set_arithmetic_flags(op1value, op2value,
result, size)

            elif op1.type == pydasm.OPERAND_TYPE_MEMORY:
                op1value = self.get_memory_address(instruction,
1, size)
                op2value = op2.immediate & self.get_mask(size)

                # Do logic
                op1valuederef = self.get_memory(op1value, size)
```

```
                    result = op1valuederef - op2value

                    self.set_arithmetic_flags(op1valuederef,
op2value, result, size)


            #83 /7 ib CMP r/m16,imm8 Compare imm8 with r/m16
            #83 /7 ib CMP r/m32,imm8 Compare imm8 with r/m32
            elif instruction.opcode == 0x83 and instruction.extindex
== 0x7:

                if so:
                    size = 2
                else:
                    size = 4

                if op1.type == pydasm.OPERAND_TYPE_REGISTER:
                    op1value = self.get_register(op1.reg, size)
                    op2value = op2.immediate & self.get_mask(size)

                    # Do logic
                    result = op1value - op2value

                    self.set_arithmetic_flags(op1value, op2value,
result, size)

                elif op1.type == pydasm.OPERAND_TYPE_MEMORY:
                    op1value = self.get_memory_address(instruction,
1, size)

                    op2value = op2.immediate & self.get_mask(size)

                    # Do logic
                    op1valuederef = self.get_memory(op1value, size)

                    result = op1valuederef - op2value

                    self.set_arithmetic_flags(op1valuederef,
op2value, result, size)

            else:
                return False

        return True
```

This verbose listing of the CMP instruction should be very readable and understandable. When PyCPU executes the mnemonic function, it then determines which opcode is requested, and based on the Intel specification, then determines which type of operand we are dealing with. This executes the logic of that opcode and sets the appropriate registers, memory, and corresponding CPU flags if necessary.

PyCPU relies on a myriad of helper functions for cleanly reading register values and memory addresses. The following synopsis helps explain the most common of the helper functions repeated throughout the source code. PyCPU relies heavily on the size

variable which determines how large the memory, or register will be. This size is then handed to all of the helper functions in an attempt to make things more universal.

- get_register(register, size)

  As can be inferred this will fetch the requested register by name or index and properly return the masked value according to the size parameter. Using this will also trigger a handler if it is defined for the register.

- set_register(register, value, size)

  Similar to get_register this function will set the register to the provided value.

- get_memory_address(instruction, operand_index, size)

  This function will calculate the address a operand is requesting. Based on the instructions ModRM/SIB byte and opcode, it will return the address being used in the operand.

- get_memory(address, size)

  This will return the value address points to in memory. Memory access will be covered in more detail later.

- set_memory(address, value, size)

  Sets the requested address to value and size.

- set_arithmetic_flags(operand_1_value, operand_2_value, result, size)

  A convenience function setting appropriate CPU flags for an arithmetic operation. These flags include CF, OF, SF, PF, and ZF

- set_shift_flags(result, size)

  Similar to set_arithmetic_flags except it updates only the flags used in a bitwise shift operation. These flags include SF, PF, and ZF

The CPU class should not be used directly. The emulator class will handle all calls to execute and memory requests. If a user wants to extend the supported instructions or behavior this would be the place to do it.

3.2 PyMemory

The second piece of the PyEmu puzzle is the memory handler. PyMemory is responsible for handling any fetches or stores of memory locations including code. This class is very basic in design as it relies heavily on user supplied functions that do the unknown memory fetching. It operates by keeping a cache of already fetched memory pages locally. If a page is not present in the cache, it will call the overloaded get_page() method. Get_page() will handle the request how the user has defined it.

```python
def get_memory(self, address, size):
    page = address & 0xfffff000
    offset = address & 0x00000fff

    # Check our cache if not fetch
    if page in self.pages:
        # Return from our cache
        rawbytes = ""
        for x in xrange(0, size):
            rawbytes += self.pages[page][offset+x]

        if size == 1:
            return struct.unpack("<B", rawbytes)[0]
        elif size == 2:
            return struct.unpack("<H", rawbytes)[0]
        elif size == 4:
            return struct.unpack("<L", rawbytes)[0]
        else:
            return rawbytes
    else:
        # We need to fetch this
        if not self.get_page(page):
            print "[!] Problem getting page"
            return False
        else:
            rawbytes = ""
            for x in xrange(0, size):
                rawbytes += self.pages[page][offset+x]

            if size == 1:
                return struct.unpack("<B", rawbytes)[0]
            elif size == 2:
                return struct.unpack("<H", rawbytes)[0]
            elif size == 4:
                return struct.unpack("<L", rawbytes)[0]
            else:
                return rawbytes

    return False
```

When in live analysis via Pydbg the get_memory() method operates as displayed below.

```python
def get_page(self, page):
    try:
```

```
            mempage = self.dbg.read_process_memory(page,
self.PAGESIZE)
        except:
            print "[!] Couldnt read mem page @ 0x%08x" % page
            return False

        self.pages[page] = mempage

        return True
```

This fetch and cache method allows PyMem to easily control where the data is coming from and how it is stored. It keeps the local copy separated from the real processes memory space. A good use for this would be for dumping all of the pages used by the emulator or restoring the emulator memory back to the debugged process.

Setting memory is very similar in operation.

```
def set_memory(self, address, value, size):
    page = address & 0xfffff000
    offset = address & 0x00000fff

    if isinstance(value, int) or isinstance(value, long):
        if size == 1:
            packedvalue = struct.pack("<B", int(value))
        elif size == 2:
            packedvalue = struct.pack("<H", int(value))
        elif size == 4:
            packedvalue = struct.pack("<L", int(value))
        else:
            print "[!] Couldnt pack new value of size %d" %
(size)

            return False
    elif isinstance(value, str):
        packedvalue = value[::-1]
    else:
        print "[!] Dont understand this value type %s" %
type(value)

        return False

    # Check our page if not fetch
    if page in self.pages:
        newpage = self.pages[page][:offset]
        for x in xrange(0, size):
            newpage += packedvalue[x]
        newpage += self.pages[page][offset + size:]

        self.pages[page] = newpage

        return True
    else:
        # We need to fetch this
        if not self.get_page(page):
            print "[!] Problem getting page"
```

```
                    return False
              else:
                  newpage = self.pages[page][:offset]
                  for x in xrange(0, size):
                       newpage += packedvalue[x]
                  newpage += self.pages[page][offset + size:]

                  self.pages[page] = newpage

                  return True

          return False
```

A powerful feature of PyMemory is the ability to implement your own memory manager. For instance, if one is so inclined, filling memory requests with "A" can be done in very few lines of code. To implement this simply overload the parent PyMemory class and provide a get_memory method for PyMemory.

```
class MyMemory(PyMemory):
    def __init__(self, fillchar="A"):
        self.fillchar = fillchar

        PyMemory.__init__(self)

    def get_page(self, page):
        try:
            mempage = self.fillchar * self.PAGESIZE
        except:
            print "[!] Couldnt read mem page @ 0x%08x" % page
            return False

        self.pages[page] = mempage

        return True
```

PyMemory should not need any external modification unless implementing a custom manager. PyMemory is extremely simple in that it gets and sets memory based on the manager you use. This is also all handled for you by the emulator class you instantiate.

3.3 PyEmu

PyEmu is the main interface between the underlying CPU and Memory classes and the user. In most cases, a user will instantiate the relevant PyEmu class and work with the provided methods in PyEmu. This layer of abstraction provides a hassle free method of operation when writing a script using PyEmu.

The exposed PyEmu methods are growing daily, and include functions to execute, query, log, debug, dump and other various ways of accessing and controlling the emulator CPU. The list below contains most of the important methods for use in PyEmu scripts and will be detailed later.

Execution:
- execute
- set_breakpoint

Modification:
- set_register
- set_stack_argument
- get_stack_argument
- get_stack_argument
- get_stack_variable
- set_stack_variable
- get_memory
- set_memory

Handlers:
- set_register_handler
- set_library_handler
- set_exception_handler
- set_instruction_handler
- set_opcode_handler
- set_memory_handler
- set_pc_handler
- set_memory_write_handler
- set_memory_read_handler
- set_memory_access_handler
- set_stack_write_handler
- set_stack_read_handler
- set_stack_access_handler
- set_heap_write_handler
- set_heap_read_handler
- set_heap_access_handler

Misc:
- log
- debug
- dump_memory
- restore_context

This base class is intended to be inherited by a more specific emulator class. For example when working in IDA Pro, a user will want to use the IDAPyEmu class as it provides the additional support needed during setup. This is not always the case and the user can of course create their own emulation class.

4 Using PyEmu

Using PyEmu should be natural and flexible. It tries to provide a logical layer to achieve goals the user may need to solve via emulation. This section will cover in more detail how it can be used. This includes creating the needed objects for emulation, setting up variables, memory, execution, and logging.

4.1 Instantiation

Instantiating a PyEmu object is the first step in creating a PyEmu script. This will create the necessary class object for everything else that is achieved in the emulator. When instantiating objects the first step is identifying what your environment will be.

There are currently three environments provided in the PyEmu package. These are presented as an interface to IDA Pro, Pydbg, and a standalone PE file. The IDA Pro interface allows a user to execute their script within IDA via IDAPython and setup necessary values for variables, arguments, and memory. The Pydbg interface lets a user writing a pydbg script use the emulator seamlessly, querying real memory and process context information such as register values and flags. The PE file represents a standalone method for utilizing an emulator without IDA, in a static analysis setting.

These classes all inherit from the base PyEmu class and are descriptively named after their environment. In the case of IDAPyEmu a user would create the object like so:

```
from PyEmu import IDAPyEmu

emu = IDAPyEmu()
```

From here the user can then access all of the exposed methods for controlling the emulator and its associated properties. The IDAPyEmu class can take many optional arguments and is defined as:

```
class IDAPyEmu(PyEmu):
    def __init__(self, stack_base=0x0095f000, stack_size=0x1000,
heap_base=0x000a0000, heap_size=0x2000, frame_pointer=True):

        self.stack_base = stack_base
        self.stack_size = stack_size
        self.heap_base = heap_base
        self.heap_size = heap_size
        self.frame_pointer = frame_pointer

        PyEmu.__init__(self)

        self.setup_memory()

    def setup_memory(self):
        # Sets up memory of emulator
        self.memory = IDAMemory()
```

```
        # Do stack initialization
        self.memory.get_page(self.stack_base)
        self.cpu.set_register32("EBP", self.stack_base -
self.stack_size / 2)
        self.cpu.set_register32("ESP",
self.cpu.get_register32("EBP") - 4)

        return True
```

4.2 Setup

Setup is necessary for populating and organizing the associated properties the emulator will need to execute as expected. This includes loading the code section and data section into memory.

For our IDAPyEmu example we would do the following using IDAPython's access to IDC.

```
textstart = SegByName(".text")
textend = SegEnd(textstart)

print "[*] Loading text section bytes into memory"

currenttext = textstart
while currenttext <= textend:
    emu.set_memory(currenttext, GetOriginalByte(currenttext),
size=1)
    currenttext += 1

print "[*] Text section loaded into memory"

datastart = SegByName(".data")
dataend = SegEnd(datastart)

print "[*] Loading data section bytes into memory"

currentdata = datastart
while currentdata <= dataend:
    emu.set_memory(currentdata, GetOriginalByte(currentdata),
size=1)
    currentdata += 1

print "[*] Data section loaded into memory"

emu.set_register("EIP", ScreenEA())
```

This will populate the code and data section at the proper base addres in the IDAMemory class, and set the PyCPU register EIP to the current address selected in IDA Pro. A few familiar methods are used to achieve this.

```
emu.set_register(register, value, name="")
```

Set register will set the indicated register to the value supplied. Differing from the PyCPU class, it can only specify the register by name. A size is not needed as it will automatically be determined based on the register name (i.e EAX, AX, AH, AL). The keyword argument is useful to set a name to the register that may make more sense to the user. For instance, the following:

```
emu.set_register("EAX", 2, name="counter")
```

Will set the ECX register to 2 and set up a name "counter" for it. This register can then be simply queried by name using get_register("counter"). Hopefully this will allow a reverse engineer easily organize their information.

```
emu.set_memory(address, value, size=1)
```

Set memory will set the value in the memory manager's cache to the provided value. An optional size argument is used because in most cases PyEmu will automatically calculate the size of the value argument. This is useful for tastslike setting string values of arbitrary length in memory.

```
emu.set_memory(0x41414141, "ABCDEFGHIJKLMNOP")
```

This example would set the memory address of 0x41414141 to the string provided, automatically calculating its length. This will also work with values of type 'long' and 'int' in which they are determined to be of 4 byte lengths. The set_memory function will then call the memory managers set_memory function.

```
def set_memory(self, address, value, size=0):
    <...>

    if not self.memory.set_memory(address, value, size):
        return False

    return True
```

This example using IDAPyEmu may seem complex at first glance. However, all we are trying to accomplish is initializing the memory and cpu for use as it would if it were to be executing on the system. Also we are telling PyCPU we want to execute from the currently selected address in our disassembly window.

4.3 Handlers

Handlers are one of the biggest benefits of using PyEmu. A handler lets a user set up certain points that need to call back into their custom code. This method of giving control to a user's script allows the user to solve some of the problems mentioned previously. PyEmu provides numerous handlers out of the box, while being designed with expansion in mind.

All of the handlers operate using function pointers. To catch the call back a user must define a function, and pass the functions name to the handler creation routine for callback when that particular situation is met. For instance

```
def my_handler(emu):
    print "[*] Hit my handler @ %x" % emu.get_register("EIP")

    return True
```

One current drawback to the handlers is that arguments are dependent on which handler you are defining. In the future this may change and be easier via a defined handler event structure passed to the user defined callback. One note is the fact all the handlers will be given an instance of the PyEmu class. This lets the script have direct access to the CPU for modification, querying, or any other tasks that need to be completed.

The following handlers are included with the PyEmu package and their associated methods are listed below.

4.3.1 Register handlers

Register handlers are as you would expect them. If the indicated register is modified, the script will receive the opportunity to act on, such as for logging the value, or modifying it.

```
emu.set_register_handler("eax", my_register_handler)
```

The register parameter mimics the set_register() method and can be used by name (i.e. EAX, AX, AL, AH) or "name" (i.e. "counter"). Register handlers are powerful when tracking modifications of a known, or important register you may want to keep an eye on.

```
def my_register_handler(emu, register, value, type)
```

The handler definition will receive an emulator object, the value of the register, and the type. Type is a string indicating a "read" or "write" of the register indicatied.

4.3.2 Library handlers

Library handlers allow a user to catch any execution of a library call before it takes place. In PyEmu, many standard library calls are emulated to provide seamless execution when calling imports. A handler can be used to change that behavior 'on the fly' for things such as controlling the location a malloc() may return.

```
emu.set_library_handler("malloc", my_library_handler)
```

The library name is the exported symbol name of the import. This is case insensitive and allows the user to tailor execution even further.

```
def my_library_handler(emu, library, address)
```

The handler definition will receive an emulator object, the name of the import being called and the address of the associated import.

4.3.3 Exception handlers

Exception handlers act as one would expect. Any time an exception is raised this function will be called. An obvious example would be catching any general protection faults due to invalid memory access.

```
emu.set_exception_handler("GP", my_exception_handler)
```

As before the first argument is the Intel fault code of the exception being thrown by the CPU.

```
def my_library_handler(emu, exception, address)
```

The handler definition will receive an emulator object, the exception thrown, and the address of the violation.

4.3.4 Instruction handlers

Instruction handlers are present to allow catching of specific mnemonics after they have been completed. Often, when reverse engineering an application certain instructions may be significant to the task. A good example of this is the "cmp" instruction used in branch decisions. If one wanted to log each "cmp" and what was being compared this would be simple using an instruction handler.

```
emu.set_instruction_handler("cmp", my_instruction_handler)
```

The handler needs only the mnemonic to be trapped on and the associated function pointer.

```
def my_cmp_handler(emu, mnemonic, op1, op2, op3)
```

The handler function will receive the emulator object, the mnemonic, and values of all the possible operands as dword integers.

4.3.5 Opcode handlers

The opcode handler is a subset of the instruction handler. This allows for more granular control over what is being accessed. If you only want to be notified when a "cmp" mnemonic is executed, but only in cases when comparing against memory as is the case with opcode 0x39.

```
emu.set_opcode_handler(0x39, my_opcode_handler)
```

Again the handler setup is simple in that it only expects the opcode you are requesting and a handler function. In the case of multi-byte opcodes simply indicate it as a int of that length (i.e. 0x0f9c)

```
def my_39_handler(emu, opcode, op1, op2, op3)
```

The handler function will receive the emulator object, the opcode, and values of all the possible operands as dword integers.

4.3.6 Memory handlers

A memory handler is provided to allow a means for catching all access to a specific address of memory. This can be either a read or write and will greatly inform the user tracking down specific memory access attempts on a known address.

```
emu.set_memory_handler(0x41424344, my_memory_handler)
```

And again we provide the dword size address of the memory we are interested in.

```
def my_memory_handler(emu, address, value, size, type)
```

The handler function will receive the emulator object, the address of the access, value being read, or written to the address, and size of the request. The type argument is a string of value "read" or "write"

4.3.6 Program counter handler

The program counter handler is used to trigger a callback when execution reaches a specified address, allowing a user to set up points in a binary allowing control to transfer back to their script.

```
emu.set_pc_handler(0x45464748, my_pc_handler)
```

Set up is the same as the rest.

```
def my_memory_handler(emu, address)
```

The handler function will receive as usual the emulator object as well as the value of the program counter register (i.e. EIP)

4.3.7 High level memory handlers

The high level memory handlers allow only one handler per action. This is provided as a simple interface to monitor memory access. These handlers monitor read, write, and access callbacks for any memory, any stack, or any heap requests from PyCPU.

```
emu.set_memory_write_handler(my_memory_write_handler)
emu.set_memory_read_handler(my_memory_read_handler)
emu.set_memory_access_handler(my_memory_access_handler)

emu.set_stack_write_handler(my_stack_write_handler)
emu.set_stack_read_handler(my_stack_read_handler)
emu.set_stack_access_handler(my_stack_access_handler)

emu.set_heap_write_handler(my_heap_write_handler)
emu.set_heap_read_handler(my_heap_read_handler)
emu.set_heap_access_handler(my_heap_access_handler)
```

There is no option to specify the address of the handler. That would be better suited for the set_memory_handler() method. Again these are convenience functions mostly for logging purposes.

```
def my_memory_write_handler(emu, address)
def my_memory_read_handler(emu, address)
def my_memory_access_handler(emu, address, type)
```

Sticking with the theme these handler functions receive an emulator object and in the case of a write or read handler the address being accessed. In the case of a memory access the type is returned containing a string of "read" or "write".

The handlers are simple to use and extremely powerful in practice. Hopefully they convey their purpose clearly and help aid in any task done with PyEmu.

4.4 Execution

Execution is the means in which the whole process of emulating code under PyEmu is driven. The basic idea is simple, we want our emulator to go from point a, to point b. This is achieved in several different ways. The execute() method is the only way of advancing the CPU and is defined as

```
execute(self, steps=1, start=0x0, end=0x0)
```

All of the arguments are optional. Used alone, it will advance based on the current program counter of PyCPU. In the case of IDAPyEmu this is the current cursor location in the disassembly. All of the optional arguments can be used in any combination and act as expected. The "steps" keyword argument defines how many instructions to be executed. Keeping an internal counter emulation ends when the steps count has been reached. Start can be specified to establish a different location for emulation than is currently present. " end" allows us to set a termination point. Note that "end" can seem misleading in a complex function as often times the address may accidentally be impossible to reach in cases where a branch or call does not return.

Execution is, and should be, simple. Giving steps, start, and ending functionality provides us 99% of the cases we may need. Again adding more is no problem.

4.5 Modification

The ability to modify and initialize data in an emulator is crucial. PyEmu tries to provide the user with the ability to specify data in various places, and organize that data so it is meaningful during reverse engineering. For most cases, data as the user sees it can be divided into 4 categories: registers, stack variables, stack arguments, and other memory. When looking at a function, these four are the biggest concern. These four categories have supporting methods for setting, and getting their values.

```
emu.set_register("eax", 0x1234567, name="counter")
emu.get_register("eax")
emu.get_register("counter")
```

The register category has been addressed before. In the case of setting a register, you must provide the name of the register being set, the value to set, and an option name for the register. Finally we see how you can access that value by name in the future. Letting us easily label information in a human readable form.

```
emu.set_stack_variable(0x80, 0x12345678, name="var_80")
emu.get_stack_variable(0x80)
emu.get_stack_variable("var_80")
```

This may seem confusing at first giving an innocuous value as the fist argument. This is simply the offset from the stack pointer or frame pointer in cases where we have a frame pointer. This is easily identified in IDA as the label of the interesting local variable. In live analysis this can be gleaned by getting an offset for the address from the pertinent stack register. The "name" optional argument allows us to organize information.

```
emu.set_stack_argument(0x8, 0xaabbccdd, name="arg_0")
emu.get_stack_argument(0x8)
emu.get_stack_argument("arg_0")
```

Similar in almost every way to the stack_variable category of methods, the stack arguments operate in the same manner, except they are the addresses of the arguments pushed on the stack before the current function frame.

```
emu.set_memory(0x12345678, "ABCDEFGHIJKLMNOP")
emu.set_memory(0x12345678, 0x12345678, size=2)
emu.get_memory(0x12345678, size=4)
```

Setting and getting other pieces of memory is straightforward. Providing an address and value, the memory address will be set to that value. Size, again, can in most cases be automatically determined, but for setting a value of differing sizes it can be provided. Get memory works as expected given an address it will dereference it and return the value. Note that if you request a string of size >4 the data will automatically be returned as a string.

5 The Real world

Now that we have a firm grasp on how everything flows and can be used, we should investigate some real world examples of using PyEmu. Various tasks have been chosen to demonstrate some obvious uses for a scriptable emulator. This should also give the reader a starting point into how they can apply PyEmu to their particular situation to maximize efficiency.

5.1 IDA Pro

We have already discussed using PyEmu in some detail. The main method currently implemented utilizes IDAPython to execute the user's script. In our script we first map the code section and data section into memory. After that has completed we can then execute the emulator and operate as intended. The following excerpt will print each instruction executing under the emulator.

```
emu = IDAPyEmu()

# Load .text and .data sections into memory
<…>

emu.set_register("EIP", ScreenEA())
emu.debug(2)

emu.execute(steps=5)
```

And when ran

```
.text:00427E6B                              ;--------------------------------------
.text:00427E6B
.text:00427E6B
.text:00427E6B                              loc_427E6B:                  ; CODE XREF:
.text:00427E6B 8B 4D E8                      mov     ecx, [ebp+var_18]
.text:00427E6E 89 4D D8                      mov     [ebp+var_28], ecx
.text:00427E71 8D 55 E8                      lea     edx, [ebp+var_18]
.text:00427E74 89 55 F4                      mov     [ebp+var_C], edx
.text:00427E77 8D 45 D8                      lea     eax, [ebp+var_28]
.text:00427E7A 83 E8 01                      sub     eax, 1
.text:00427E7D 89 45 F0                      mov     [ebp+arg_18], eax
.text:00427E80 C7 45 F8 04 00+               mov     [ebp+var_8], 4
.text:00427E87
.text:00427E87                              loc_427E87:                  ; CODE XREF:
.text:00427E87 8B 4D F8                      mov     ecx, [ebp+var_8]
.text:00427E8A 83 E9 01                      sub     ecx, 1
.text:00427E8D 89 4D F8                      mov     [ebp+var_8], ecx
```

```
[*] Loading text section bytes into memory
[*] Text section loaded into memory
[*] Loading data section bytes into memory
[*] Data section loaded into memory
[*] Executing [0x427e6b][8b] mov ecx,[ebp-0x18]
[*] Executing [0x427e6e][89] mov [ebp-0x28],ecx
[*] Executing [0x427e71][8d] lea edx,[ebp-0x18]
[*] Executing [0x427e74][89] mov [ebp-0xc],edx
[*] Executing [0x427e77][8d] lea eax,[ebp-0x28]
[*] Executing [0x427e7a][83] sub eax,0x1
[*] Executing [0x427e7d][89] mov [ebp-0x10],eax
[*] Executing [0x427e80][c7] mov dword [ebp-0x8],0x4
[*] Done
```

5.2 Pydbg

Pydbg can also be a vehicle for emulation tasks. This live analysis option can provide powerful flexibility in determining code paths and application behavior. The addition of the emulator should seamless. In this example, a script will set up the pydbg instance and attach to a running process of the users choosing. Once attached, a break point is set indicating the address we want to start emulation. When the address is hit, we start a loop using the emulator as a console to inspect register values. A snippet of the script looks like this.

```python
# Our pydbg initial break point handler
def handler_breakpoint(dbg):
    # Initial module bp we need to process entries
    if dbg.first_breakpoint:
        print "[*] First bp hit setting emu address @ 0%08x" %
dbg.emuaddress

        # Set up a custom break point handler for the emulator
        dbg.bp_set(dbg.emuaddress,
handler=handler_emu_breakpoint, restore=False)

        return DBG_CONTINUE

    print "[!] Unknown bp caught @ 0%08x" % dbg.exception_address

    return DBG_CONTINUE

# Do the emulation once the requested bp has been reached
def handler_emu_breakpoint(dbg):
    if dbg.exception_address != dbg.emuaddress:
        print "[!] Emulator handler caught unknown bp @ 0x%08x" %
(dbg.exception_address)
```

```python
        return DBG_CONTINUE

    # Create a new emulator object passing a pydbg instance
    emu = PyDbgPyEmu(dbg)

    c = None
    while c != "x":
        emu.execute()
        emu.dump_regs()

        c = raw_input("emulator> ")

    return DBG_CONTINUE

procname = sys.argv[1]
emuaddress = sys.argv[2]

dbg.set_callback(EXCEPTION_BREAKPOINT, handler_breakpoint)

if not attach_target_proc(dbg, procname):
    print "[!] Couldnt load/attach to %s" % procname

    sys.exit(-1)

dbg.debug_event_loop()
```

Once executed, the output can be seen as:

```
C:\Code\Python\PyEmu\examples>pydbgpyemu.py calc.exe 0x001001AF3

[*] Trying to attach to existing calc.exe
[*] Attaching to calc.exe (2516)
[*] First bp hit setting emu address @ 001001af3

[*] Executing [0x1001af3][6a] push byte 0xc
EAX: 0x0000002e
ECX: 0x00000000
EDX: 0x00000005
EBX: 0x010012a0
EBP: 0x0007f99c
ESP: 0x0007f90c
ESI: 0x00000001
EDI: 0x0007f95c
EFLAGS: 0x244 [ ZF PF IF ]
EIP: 0x01001af5
emulator> t

[*] Executing [0x1001af5][58] pop eax
EAX: 0x0000000c
ECX: 0x00000000
EDX: 0x00000005
EBX: 0x010012a0
EBP: 0x0007f99c
ESP: 0x0007f910
ESI: 0x00000001
EDI: 0x0007f95c
```

```
EFLAGS: 0x244 [ ZF PF IF ]
EIP: 0x01001af6
emulator> t

[*] Executing [0x1001af6][33] xor edi,edi
EAX: 0x0000000c
ECX: 0x00000000
EDX: 0x00000005
EBX: 0x010012a0
EBP: 0x0007f99c
ESP: 0x0007f910
ESI: 0x00000001
EDI: 0x00000000
EFLAGS: 0x240 [ ZF IF ]
EIP: 0x01001af8
emulator> t

[*] Executing [0x1001af8][57] push edi
EAX: 0x0000000c
ECX: 0x00000000
EDX: 0x00000005
EBX: 0x010012a0
EBP: 0x0007f99c
ESP: 0x0007f90c
ESI: 0x00000001
EDI: 0x00000000
EFLAGS: 0x240 [ ZF IF ]
EIP: 0x01001af9
emulator> x

[*]Exiting the emulator.

C:\Code\Python\PyEmu>
```

This snippet of code is very simple and straightforward. With an enhanced emulator console, the bridge between live execution and emulated execution could be realized.

5.3 PE

The PE file format contains all the necessary information for running an application. This includes the various sections of code, data, and their associated relative addresses from the image base. Since we have access to this information and the pefile python library, a quick implementation of a PEPyEmu class is complete. This class allows you to write scripts without the need for IDA's disassembly. The script to use this is simple. It takes an executable name and address, emulating for 10 steps.

```
#!/usr/bin/env python

import os, sys, pefile

from PyEmu import PEPyEmu

exename = sys.argv[1]
address = int(sys.argv[2], 16)
```

```
emu = PEPyEmu(exename)
emu.debug(2)

emu.set_register("EIP", address)

emu.execute(steps=10)
```

And the output from the script

```
C:\Code\Python\PyEmu>pepyemu.py "examples\calc.exe" 0x010022F9

[*] Image Base Addr:  0x01000000
[*] Code Base Addr:   0x01001000
[*] Data Base Addr:   0x01014000
[*] Entry Point Addr: 0x01012475

[*] Loading text section bytes into memory
[*] Text section loaded into memory
[*] Loading data section bytes into memory
[*] Data section loaded into memory

[*] Executing [0x10022f9][55] push ebp
[*] Executing [0x10022fa][8b] mov ebp,esp
[*] Executing [0x10022fc][81] sub esp,0x108
[*] Executing [0x1002302][53] push ebx
[*] Executing [0x1002303][56] push esi
[*] Executing [0x1002304][8b] mov esi,[ebp+0xc]
[*] Executing [0x1002307][8b] mov eax,[esi+0x10]
[*] Executing [0x100230a][57] push edi
[*] Executing [0x100230b][33] xor edi,edi
[*] Executing [0x100230d][89] mov [esi+eax*2+0x14],di

C:\Code\Python\PyEmu>
```

This example demonstrates the flexibility of PyEmu. Since the only requirement is raw bytes of instructions, the possibilities for application are numerous. This can be achieved because PyEmu strives to be as autonomous as possible when dealing with implemented PyEmu classes. By doing this, we allow the user to have full control over what they are trying to achieve. It would even be possible to create a NetPyEmu if so desired.

5.4 Tracking memory access

Determining when memory is being read and written to is crucial in understanding how an application is working. An often asked question when determining this is "When and where is memory being accessed". To solve this with PyEmu, we can set up some higher level memory access handlers. These handlers will return control when something modifies process memory. The following example is used in IDA Pro.

```
from PyEmu import IDAPyEmu

def my_memory_access_handler(emu, address, value, size, type):
```

```
    print "[*] Hit my_memory_access_handler %x: %s (%x, %x, %x,
%s)" % (emu.get_register("EIP"), emu.get_disasm(), address,
value, size, type)

    return True

# Our usual IDA setup mapping relevant sections
<…>

# Start the program counter at the current location in the
disassembly window
emu.set_register("EIP", ScreenEA())

# Set up our memory access handler
emu.set_memory_access_handler(my_memory_access_handler)

emu.execute(start=0x00427E6B, end=0x00427E8D)

print "[*] Done"
```

And the output is below

Solving a problem and answering questions like this aid the reverse engineer in accelerating up the understanding of a function, or group of functions. We also see the use of executing from start and end method for quickly defining a bounds in the emulator.

5.5 Path enumeration

Previously we demonstrated an extremely complex function. The function included hundreds of code path decisions and appears as a spider web of branches and loops. To alleviate this, one can use PyEmu to track those branches, their conditions and the values used in the decision. In this case, hooking each call to the mnemonic "cmp" provides us a simple view of the comparisons happening before each branch is taken. While this can be done in other ways we might also want to provide specific values to change the code path. In IDAPyEmu we would simply set up a mnemonic handler for "cmp" and log its values.

```
from PyEmu import IDAPyEmu

def my_cmp_handler(emu, address, op1, op2, op3):
    print "[*] Hit my_cmp_handler %x: %s (%x, %x)" %
(emu.get_register("EIP"), emu.get_disasm(), op1, op2)

    return True

# Start the program counter at the current location in the
disassembly window
emu.set_register("EIP", ScreenEA())

# This demonstrates setting local variables used in our
comparisons
emu.set_stack_variable(0x2c, 0x00000000, name="var_2C")
emu.set_stack_variable(0x1d, 0x00000001, name="var_1D")
emu.set_stack_variable(0x1e, 0x00000002, name="var_1E")

# Set up our memory access handler
emu.set_mnemonic_handler("cmp", my_cmp_handler)

emu.execute(start=0x00427E46, end=0x00427E6B)

print "[*] Done"
```

This script would result in the following

```
 * .text:00427E43 83 C4 08                               add     esp, 8
 * .text:00427E46 89 45 D4                               mov     [ebp+var_2C], eax
 * .text:00427E49 83 7D D4 00                            cmp     [ebp+var_2C], 0
   .text:00427E4D 74 08                                  jz      short loc_427E57
   .text:00427E4F 8B 45 D4                               mov     eax, [ebp+var_2C]
 * .text:00427E52 E9 CA 00 00 00                         jmp     loc_427F21
   .text:00427E57                      ; ---------------------------------------------
   .text:00427E57
   .text:00427E57                      loc_427E57:                     ; CODE XREF: sub_427E
 * .text:00427E57 0F B6 55 E3                            movzx   edx, [ebp+var_1D]
 * .text:00427E5B 0F B6 45 E2                            movzx   eax, [ebp+var_1E]
 * .text:00427E5F 3B D0                                  cmp     edx, eax
   .text:00427E61 7C 08                                  jl      short loc_427E6B
 * .text:00427E63 83 C8 FF                               or      eax, 0FFFFFFFFh
 * .text:00427E66 E9 B6 00 00 00                         jmp     loc_427F21
   .text:00427E6B                      ; ---------------------------------------------
   .text:00427E6B
   .text:00427E6B                      loc_427E6B:                     ; CODE XREF: sub_427E
   .text:00427E6B 8B 4D E8                               mov     ecx, [ebp+var_18]

[*] Loading text section bytes into memory
[*] Text section loaded into memory
[*] Loading data section bytes into memory
[*] Data section loaded into memory
[*] Hit my_cmp_handler 427e49: cmp dword [ebp-0x2c],0x0 (0, 0)
[*] Hit my_cmp_handler 427e5f: cmp edx,eax (0, 2)
[*] Done
```

5.6 Function return value statistics

Functions are often used for simple purposes. One might have a function calculating values based on input. This can be easily gathered via emulation. The concept is to set up a list of inputs, and retrieve the return value once sent through a function. This can be done as many times as needed to determine what might be the result of a function.

The simple example we will write, set up function arguments, and hook ret so that when the function ends we can log the result and start again.

```
from PyEmu import IDAPyEmu

def reset_stack(emu, value1, value2, value3):
    emu.set_stack_argument(0x8, value1, name="arg_0")
    emu.set_stack_argument(0xc, value2, name="arg_4")
    emu.set_stack_argument(0x10, value3, name="arg_8")

    return True
```

This function will reset our stack variables to their intended values.

```
def my_ret_handler(emu, address):
    global count

    value1 = emu.get_stack_argument("arg_0")
    value2 = emu.get_stack_argument("arg_4")
    value3 = emu.get_stack_argument("arg_8")

    print "[*] Returning %x: %x, %x, %x = %x" % (address, value1,
value2, value3, emu.get_register("EAX"))

    reset_stack(emu, value1 + 1, value2 + 2, value3 + 3)
    emu.set_register("EIP", ScreenEA())

    count += 1
```

```
        return True
```

Our "ret" mnemonic handler will be called upon return. When hit, we will get the value of stack arguments and the return value of the function for logging purposes. After we have logged the requested information, we increment the values, reset the program counter and do it again.

```
# Typical ida loading
<…>

# This sets our stack values for the function
reset_stack(emu, 0x00000000, 0x00000001, 0x00000002)

# Set up our memory access handler
emu.set_mnemonic_handler("ret", my_ret_handler)

count = 0
while count <= 10:
    if not emu.execute():
        break

print "[*] Done"
```

After 10 iterations of the function have been completed the emulator will exit. And here is the output.


```
.text:0041DC40 55                                 push    ebp
.text:0041DC41 8B EC                              mov     ebp, esp
.text:0041DC43 8B 45 08                           mov     eax, [ebp+arg_0]
.text:0041DC46 23 45 0C                           and     eax, [ebp+arg_4]
.text:0041DC49 8B 4D 08                           mov     ecx, [ebp+arg_0]
.text:0041DC4C 23 4D 10                           and     ecx, [ebp+arg_8]
.text:0041DC4F 0B C1                              or      eax, ecx
.text:0041DC51 8B 55 0C                           mov     edx, [ebp+arg_4]
.text:0041DC54 23 55 10                           and     edx, [ebp+arg_8]
.text:0041DC57 0B C2                              or      eax, edx
.text:0041DC59 5D                                 pop     ebp
.text:0041DC5A C3                                 retn
.text:0041DC5A                 sub_41DC40         endp
.text:0041DC5A
.text:0041DC5A                            ; ------------------------------------------
.text:0041DC5B CC CC CC CC CC                      align 10h
.text:0041DC60
.text:0041DC60                            ; !!!!!!!!!!!!!!!! S U B R O U T I N E !!!!!!!!!!
.text:0041DC60
.text:0041DC60                            ; Attributes: bp-based frame
.text:0041DC60

[*] Loading text section bytes into memory
[*] Text section loaded into memory
[*] Loading data section bytes into memory
[*] Data section loaded into memory
[*] Returning 41dc5a: 0, 1, 2 = 0
[*] Returning 41dc5a: 1, 3, 5 = 1
[*] Returning 41dc5a: 2, 5, 8 = 0
[*] Returning 41dc5a: 3, 7, b = 3
[*] Returning 41dc5a: 4, 9, e = c
[*] Returning 41dc5a: 5, b, 11 = 1
[*] Returning 41dc5a: 6, d, 14 = 4
[*] Returning 41dc5a: 7, f, 17 = 7
[*] Returning 41dc5a: 8, 11, 1a = 18
[*] Returning 41dc5a: 9, 13, 1d = 19
[*] Returning 41dc5a: a, 15, 20 = 0
[*] Done
```

Like all of our examples, this can prove useful in situations when functions may return important values that are unknown. We are aiming for reduction of time investment in each function while reverse engineering.

We have seen a few real world implementations and uses for PyEmu. There are numerous possibilities when reverse engineering and hopefully this has demonstrated some basic ones while working to create more complex solutions for your own specific needs.

6 Limitations and future work

Obviously, there are several limitations in the current toolset of reverse engineering and PyEmu. There is still a lot of manual interaction and setup when using PyEmu. Setting memory values, updating stack variables and the basic need to have some understanding of the emulated code is a draw back to any modern emulator. As the tool matures, these issues will hopefully be addressed. Whether this is done through pre-analysis, statistics, or artificial intelligence is unknown. Regardless, in order to reach the goal of reducing our time investment in reverse engineering these advancements must be made.

The lack of peripheral device emulation may also have negative side effects in PyEmu. Often times deep complex code paths may make an attempt to access a peripheral. In this case the emulator will be forced to ignore any access and continue on as if nothing happened. In the future, these cases may be rectified by having more intelligent responses to unsupported actions, such as emulating an input device.

The single biggest drawback to the current PyEmu emulator is the lack of a complete set of emulated libraries and system calls. All programs will import several external libraries for use during execution. For a library call, this may not be a large concern. In future releases, PyEmu will load the requested library into memory and provide access to its exports as is normally done when executing. However, system calls are fairly hard to emulate at this level. Although PyEmu does a decent job of attempting to provide a python usable equivalent to a socket, for instance, many other actions will go ignored. Hopefully, a decent solution for this will materialize very soon.

In the future, PyEmu will be much more automated, or at least have automation added to the base for use. Also, better library and system call support will raise the emulator to a new level. With this in mind, it still functions well and is a valid solution to most of today's reverse engineering tasks.

7 Conclusion

Emulation has played a key role in advancing computer science since the mid 1960s. As we move towards advancing reverse engineering, I believe it is beneficial to also allow emulators to demonstrate their usefulness in the field. With increasingly complex

applications, obfuscation, and never ending time constraints we must work faster and more efficiently.

PyEmu was designed with all of this in mind and most importantly to be usable, flexible, and easily extended. PyEmu strives to work fluently and as expected so that it may be integrated with the ever growing tool box of reverse engineers. Hopefully it accomplishes all of that and then some.

References

1. The History of Emulation
   http://www.zophar.net/articles/art_14-2.html

2. bochs: The Open Source IA-32 Emulation Project
   http://bochs.sourceforge.net/

3. IDA Pro Disassembler
   http://datarescue.com/idabase/index.htm

4. SABRE Security BinNavi
   http://www.sabre-security.com/products/binnavi.htm

5. PaiMei Reverse Engineering Framework
   http://paimei.openrce.org/

6. The x86 Emulator plugin for IDAPro
   http://ida-x86emu.sourceforge.net/

7. IDAPython
   http://d-dome.net/idapython/

8. PIDA
   http://paimei.openrce.org/

9. Pydbg
   http://paimei.openrce.org/

10. pefile
    http://dkbza.org/pefile.htm

11. pydasm
    http://dkbza.org/pydasm.html

12. libdasm
    http://www.nologin.org/main.pl?action=codeView&codeId=49&

13. Intel 64 and IA-32 Architectures Software Developer's Manual
    http://www.intel.com/design/processor/manuals/253666.pdf