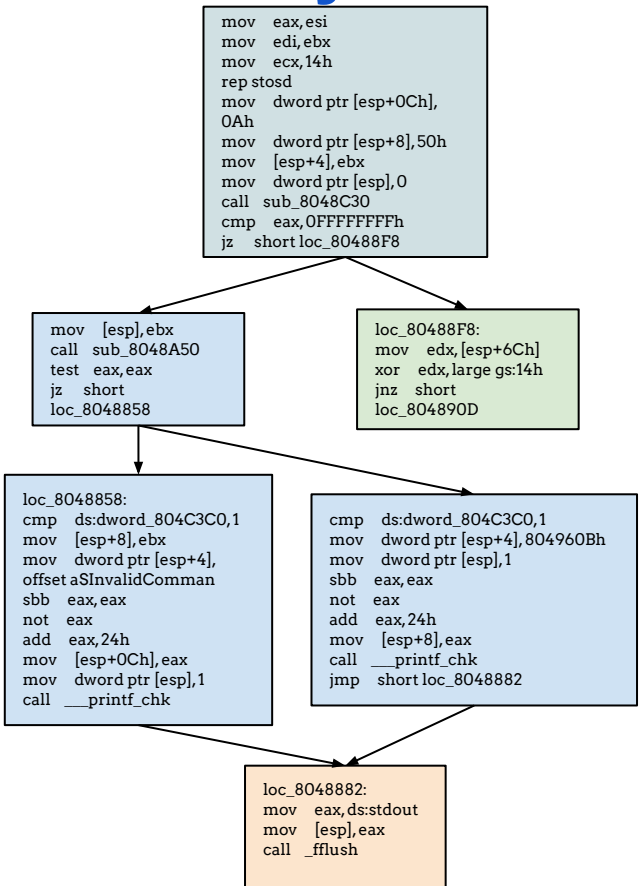
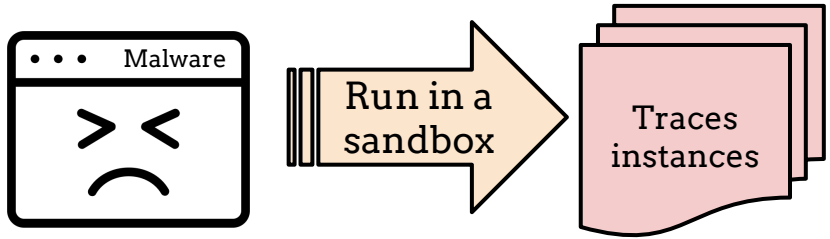
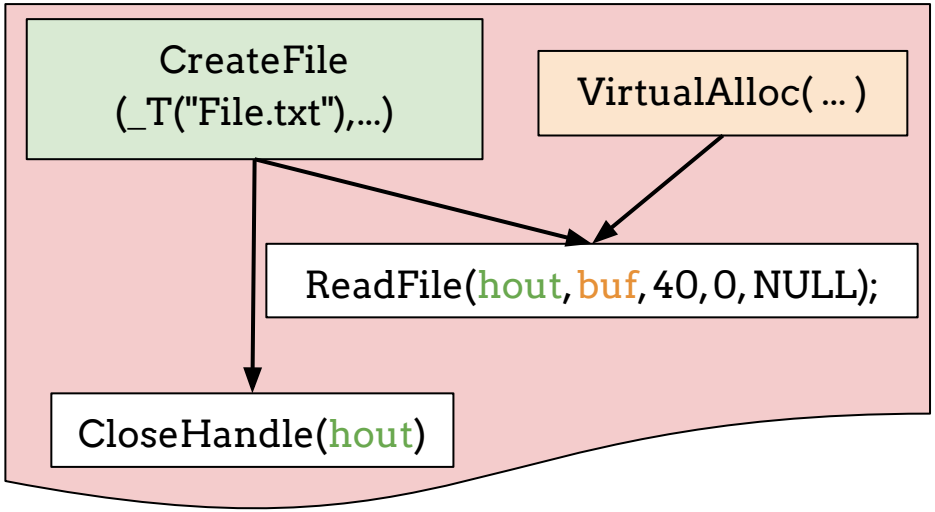
Malware Analysis

Static n Dynamic Tools

Static vs Dynamic Analysis







Static vs Dynamic Analysis

Static

Pros: high code coverage

Cons: obfuscation, labor-intensive

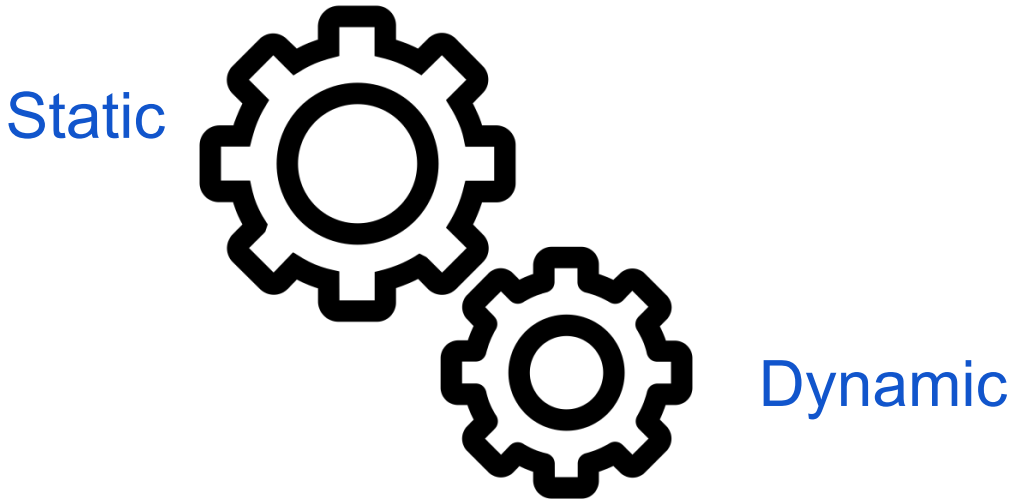
Dynamic

Pros: no semantic gap to be reversed with human skills

Cons: low code coverage, can be evaded

Static and Dynamic Analysis

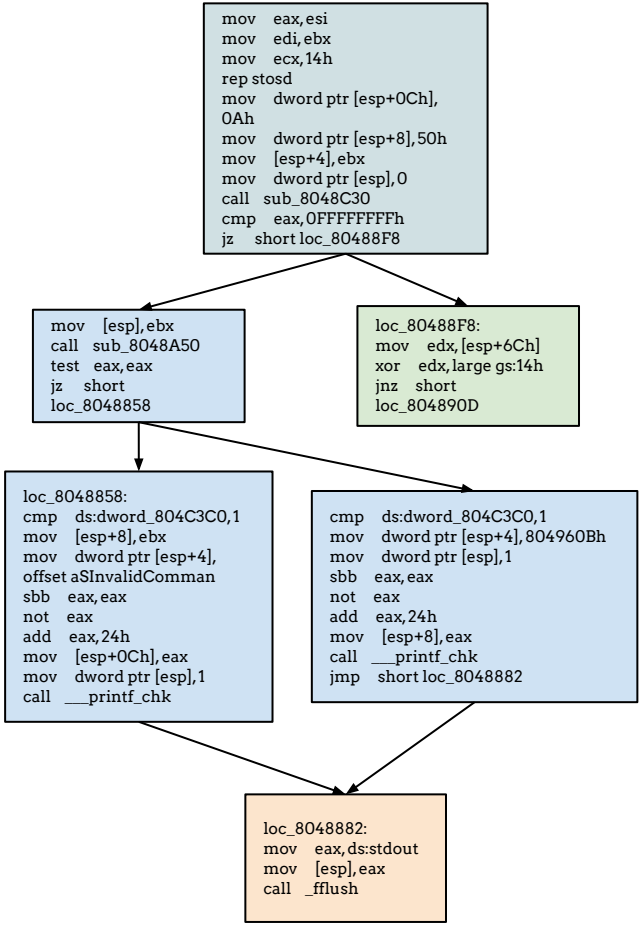
The reverse engineering process is a **sequence** of **static** and **dynamic** analysis that slowly **refine** your knowledge about the malware sample.



Static Analysis

Understand the functionalities of a binary looking at its code.

- Disassemble instructions
- Recover CFG
- Recover Functions
- Recover Types
- Decompile
- Fingerprints



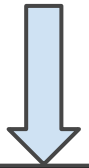
High-level and Machine Code

Static Analysis - Disassembler

- Linear Sweep

- Recursive

```
0000000: 01111111 01000101 01001100 01000110 00000001 00000001
0000006: 00000001 00000000 00000000 00000000 00000000 00000000
000000c: 00000000 00000000 00000000 00000000 00000010 00000000
0000012: 00000011 00000000 00000001 00000000 00000000 00000000
0000018: 11000000 10000011 00000100 00001000 00110100 00000000
000001e: 00000000 00000000 10110100 00001100 00000000 00000000
0000024: 00000000 00000000 00000000 00000000 00110100 00000000
000002a: 00100000 00000000 00001000 00000000 00101000 00000000
```



```
push   %ebp
mov    %esp,%ebp
and    $0xfffffff0,%esp
sub    $0x20,%esp
mov    0xc(%ebp),%eax
add    $0x4,%eax
mov    (%eax),%eax
mov    %eax,(%esp)
call   80483b0 <atoi@plt>
mov    %eax,0x1c(%esp)
mov    0xc(%ebp),%eax
```

Static Analysis - Disassembler Tools

objdump - Disasm

radare2 - Disasm

capstone - **Programmable** Disasm

Binary Ninja - Disasm + Primitive Decompiler

GHIDRA - Disasm + Decompiler (**https://ghidra-sre.org/**)

IDA Pro - Disasm + Decompiler (de facto standard)

Static Analysis - Decompile

You can go back to pseudo code:

```
push   %ebp
mov    %esp,%ebp
and    $0xfffffff0,%esp
sub    $0x20,%esp
mov    0xc(%ebp),%eax
add    $0x4,%eax
mov    (%eax),%eax
mov    %eax,(%esp)
call   80483b0 <atoi@plt>
mov    %eax,0x1c(%esp)
mov    0xc(%ebp),%eax
```

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int32_t foo(int32_t a, int32_t b);

// From module:   layout.c
// Address range: 0x80484ac - 0x80484cd
// Line range:    5 - 10
int32_t foo(int32_t a, int32_t b) {
    int32_t c = 14 * (b + a); // 0x80484c4
    return c;
}

// From module:   layout.c
// Address range: 0x80484cf - 0x8048559
// Line range:    13 - 30
int main(int argc, char **argv) {
    int32_t apple = (int32_t)argv; // 0x80484d8
    int32_t str_as_i = atoi((int8_t *)*(int32_t *)(apple + 4));
    int32_t str_as_i2 = atoi((int8_t *)*(int32_t *)(apple + 8));
    int32_t banana = foo(str_as_i, str_as_i2); // 0x804850f
    gets(NULL);
    puts(NULL);
    printf("foo(%d, %d) = %d\n", str_as_i, str_as_i2, banana);
    return 0;
}
```

Static Analysis - Decompile

Binary Ninja - Disasm + Primitive Decompiler

GHIDRA - Disasm + Decompiler (**https://ghidra-sre.org/**)

IDA Pro - Disasm + Decompiler (de facto standard)

rev.ng - not available yet

Static Analysis - Struct

Building correct types make decompilation more readable.

```
void __fastcall sub_1413(__int64 a1)
{
  signed int i; // [rsp+10h] [rbp-10h]
  signed int v2; // [rsp+14h] [rbp-Ch]
  void *v3; // [rsp+18h] [rbp-8h]

  for ( i = 0; i <= 15; ++i )
  {
    if ( !*(_DWORD *)(24LL * i + a1) )
    {
      printf("Size: ");
      v2 = sub_1AD5();
      if ( v2 > 0 && v2 <= 88 )
      {
        v3 = calloc(v2, 1uLL);
        if ( !v3 )
          exit(-1);
        *(_DWORD *)(24LL * i + a1) = 1;
        *(_QWORD *)(a1 + 24LL * i + 8) = v2;
        *(_QWORD *)(a1 + 24LL * i + 16) = v3;
        printf("Chunk %d Allocated\n", (unsigned int)i);
      }
      else
      {
        puts("Invalid Size");
      }
      return;
    }
  }
}
```

```
void __fastcall allocate(nota *notes)
{
  signed int i; // [rsp+10h] [rbp-10h]
  signed int sz; // [rsp+14h] [rbp-Ch]
  void *chunk; // [rsp+18h] [rbp-8h]

  for ( i = 0; i <= 15; ++i )
  {
    if ( !notes[i].state )
    {
      printf("Size: ");
      sz = get_long();
      if ( sz > 0 && sz <= 0x58 )
      {
        chunk = calloc(sz, 1uLL);
        if ( !chunk )
          exit(-1);
        notes[i].state = 1;
        notes[i].size = sz;
        notes[i].data = (__int64)chunk;
        printf("Chunk %d Allocated\n", (unsigned int)i);
      }
      else
      {
        puts("Invalid Size");
      }
      return;
    }
  }
}
```

```
  signed int i; // [rsp+10h] [rbp-10h]
  signed int v2; // [rsp+14h] [rbp-Ch]
  void *v3; // [rsp+18h] [rbp-8h]

  for ( i = 0; i <= 15; ++i )
  {
    if ( !*(_DWORD *)(24LL * i + a1) )
    {
      printf("Size: ");
      v2 = sub_1AD5();
      if ( v2 > 0 && v2 <= 88 )
      {
        v3 = calloc(v2, 1uLL);
        if ( !v3 )
          exit(-1);
        *(_DWORD *)(24LL * i + a1) = 1;
        *(_QWORD *)(a1 + 24LL * i + 8) = v2;
        *(_QWORD *)(a1 + 24LL * i + 16) = v3;
        printf("Chunk %d Allocated\n", (unsigned
int)i);
      }
      else
      {
        puts("Invalid Size");
      }
      return;
    }
  }
```

```
  signed int i; // [rsp+10h] [rbp-10h]
  signed int sz; // [rsp+14h] [rbp-Ch]
  void *chunk; // [rsp+18h] [rbp-8h]

  for ( i = 0; i <= 15; ++i )
  {
    if ( !notes[i].state )
    {
      printf("Size: ");
      sz = get_long();
      if ( sz > 0 && sz <= 0x58 )
      {
        chunk = calloc(sz, 1uLL);
        if ( !chunk )
          exit(-1);
        notes[i].state = 1;
        notes[i].size = sz;
        notes[i].data = (__int64)chunk;
        printf("Chunk %d Allocated\n",
(unsigned int)i);
      }
      else
      {
        puts("Invalid Size");
      }
      return;
    }
  }
```

Static Analysis - Struct IDA (local type)

Building correct types make decompilation more readable.

```
struct __attribute__((aligned(8))) nota
{
  int state;
  int unk1;
  __int64 size;
  __int64 data;
};
```

Static Analysis - Other Tools (Lifter)

You can Lift binary to perform program analysis tasks.

Angr - Binary Analysis (VEX IR) + Symbolic Execution (http://angr.io/)

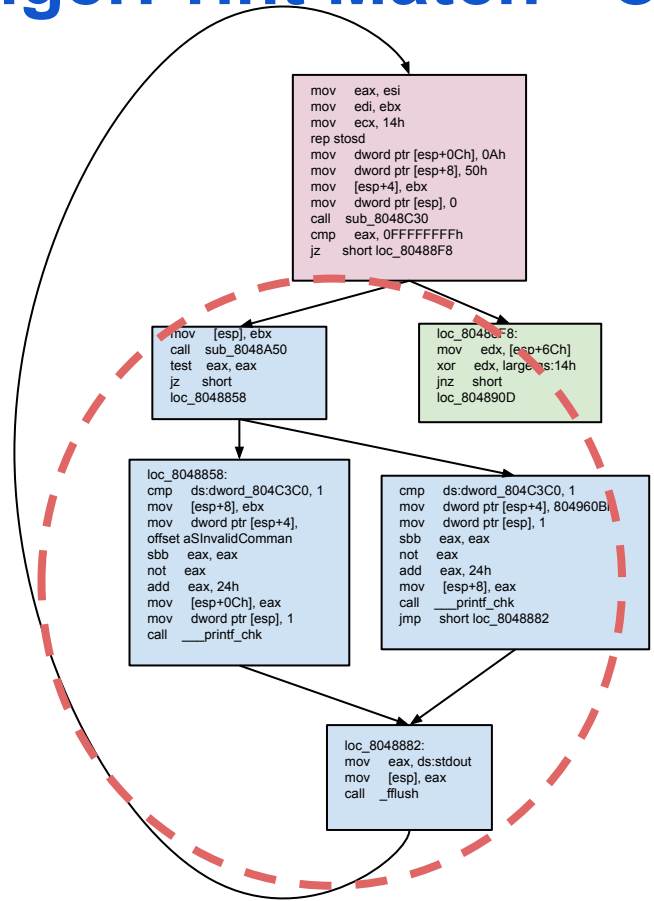
rev.ng - Binary analysis with LLVM IR + Binary translationtion

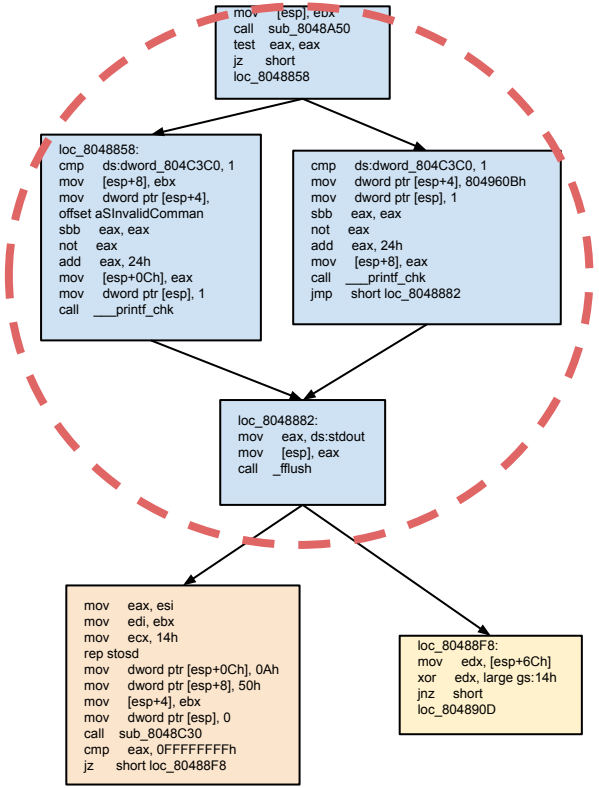
BAP - Binary analysis with BIL IR

FingerPrint Match - YARA Rule

```
rule silent_banker : banker
{
    meta:
        description = "This is just an example"
        thread_level = 3
        in_the_wild = true
    strings:
        $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
        $b = {8D 4D B0 2B C1 83 C0 27 99 6A ?? ?? F7 F9}
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT" nocase wide
        $d = {FE 39 45 [0-8] 89 00}
        $re1 = /md5: [0-9a-fA-F]{32}/
    condition:
        $a or $b and (#c > 5)
}
```

FingerPrint Match - Complex fingerprints



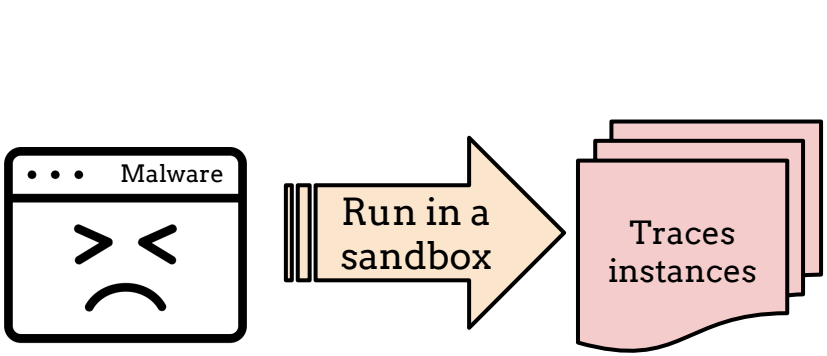


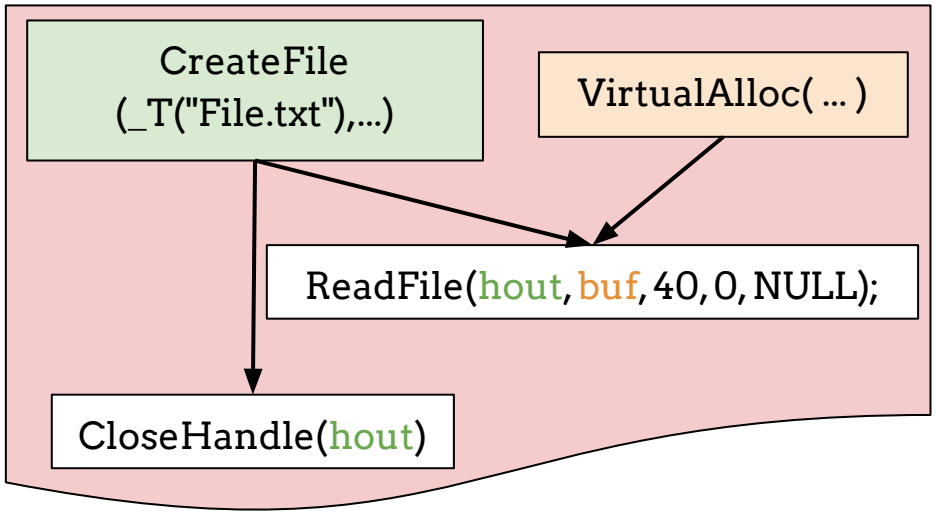
Static Analysis Problem

- Obfuscation
  - movfuscator
- Packing
  - Compress/Decompress
  - Encrypt/Decrypt
- Metamorphic components

Intro Dynamic Malware Analysis

Run the malware sample see what it does.





Things you can look at

Memory

Syscall

Network

Disk

Things you can look at: Memory

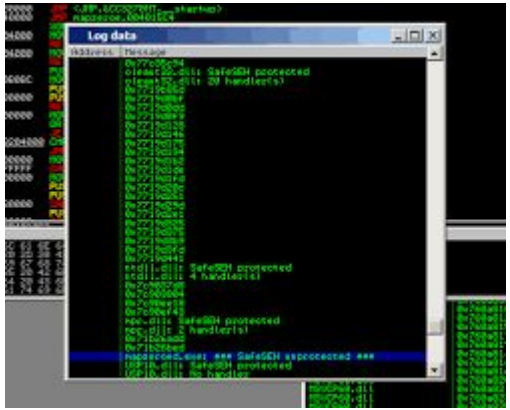
- **Debugger**
  - Run the executable and see/modify process state live.

- **Memory Forensics**:
  - Look at Infected Machine Memory.

Debuggers

- Run the executable and see/modify process state live.
- Step by step Debugging

- BreakPoints
  - **Hardware breakpoints**
- **Watchpoints**
- syscall catch
- signal catch
- **Scriptable**



- Tools: windbg, ollydbg, Immunity dbg, IDA pro, GDB, etc.

Memory Forensics

- Machine Memory (nGB)
- Process (mkB)

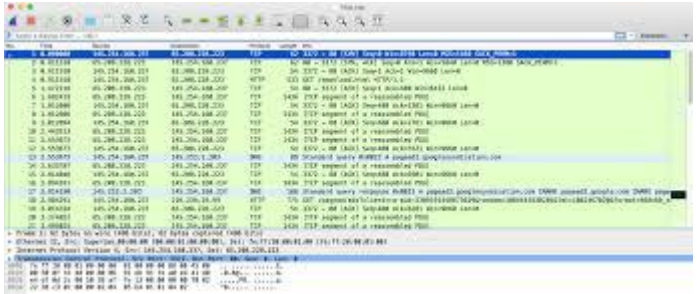
- Kernel Structure to reconstruct process memory (**OS Dependant**)
- Reconstruct **Virtual Memory** of processes.
- **List Processes** (PEB List, or Headers)
- Find CodeInjection, API Hooks, Windows Regs, etc.

Tools:

- rekall (https://github.com/google/rekall)
- volatility(https://github.com/volatilityfoundation/volatility)

Things you can look at: Network

- **Syscall**
  - Intercept all network syscall



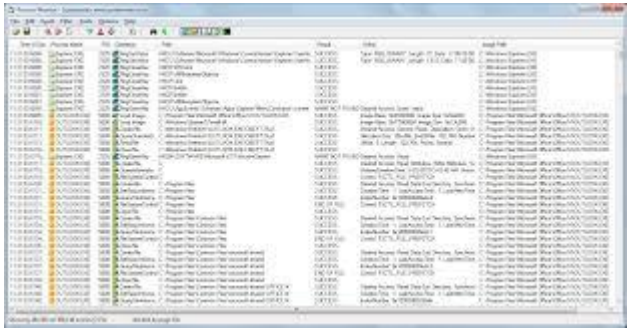
- **Network Monitor**:
  - Watch what is coming from the cable
  - Tools:
    - wireshark
    - tcpdump
    - mitmproxy
    - ngrep

Things you can look at: Disk

- **Syscall**
  - **Intercept** all disk syscall (CreateFile, Read, Write, etc)

- **Filesystem**:
  - Watch what is going to the disk
    - **MiniFilter**
  - You can monitor SATA protocol on physical machine (Lo-Phi)

Things you can look at: Syscall/API



- **Syscall Monitor**
  - Linux/MacOSx:
    - ltrace, strace
  - Windows:
    - Process Monitor (https://docs.microsoft.com/en-us/sysinternals/downloads/process-utilities)

- **Syscall/API Hook**:
  - Patch library Function to log when they are called.
    - cuckoomon, arancino, etc.

Where to look

Process

System

Machine

- **VM**
- **Bare Metal**

Where to look: Process

- Debuggers
- Syscall/API
- Dynamic Binary Instrumentation:
  - Intel Pin
  - DynamoRio (https://www.dynamorio.org/)
  - rev.ng (https://rev.ng/)

Where to look: System

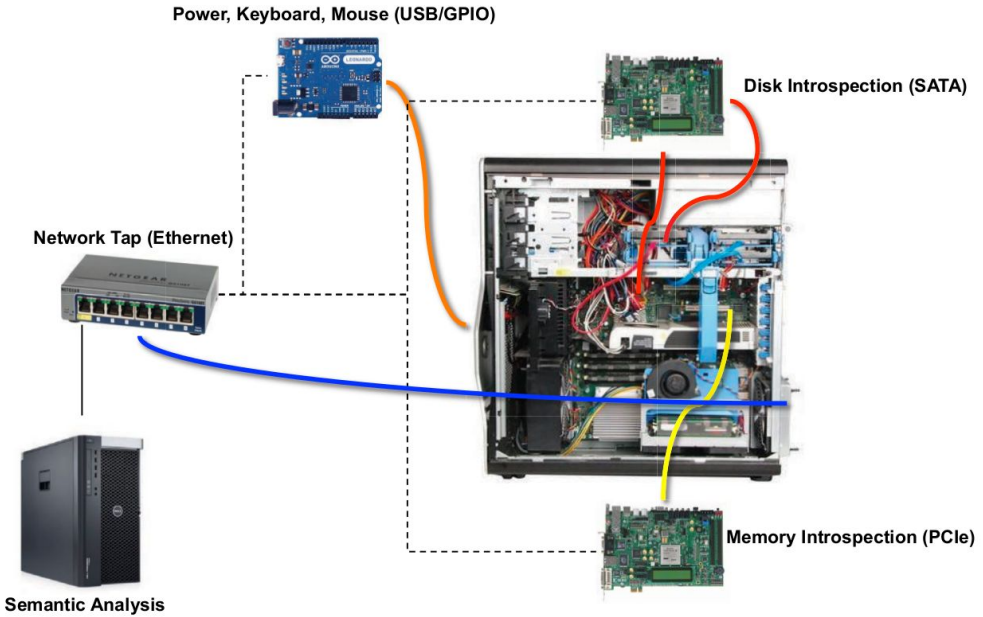
- Instrument the kernel
  - Linux
  - Windows, MacOSX
- Kernel Debug
- **Instrumenting Agent**
  - cuckoo (https://cuckoosandbox.org/)
- Drivers run into the kernel

Where to look: Machines (VM)

- Monitor Perimeter
  - Network
- Instrumented **Emulator**
  - PANDA (https://github.com/panda-re/panda) Taint
- Instrumented **HyperVisor**
  - drakvuf (https://drakvuf.com/) Stealth

Where to look: Machines (Bare Metal)

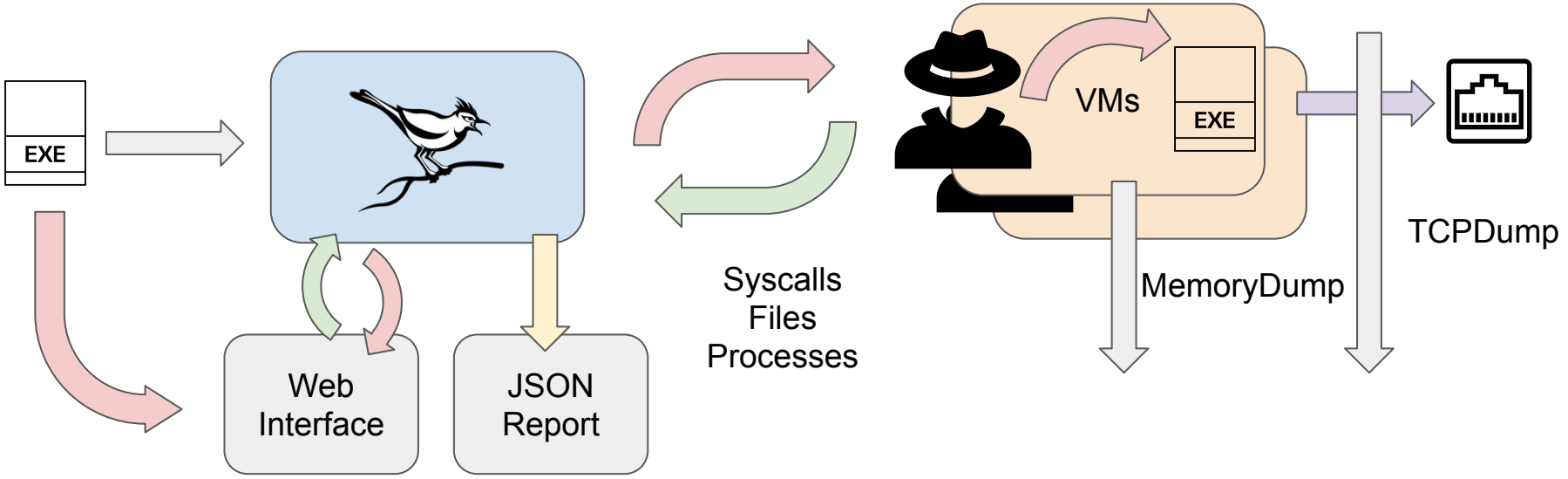
- SATA Monitor
- PCI Monitor (Memory)
- Network
- USB



Run Malware into an Instrumented Sandbox

- You can configure all the tools we just saw to be always available.
- Free Sandboxes for Malware Analysis:
  - Cuckoo (https://cuckoosandbox.org/)
  - drakvuf (https://drakvuf.com/)

Cuckoo Sandbox



Cuckoo Feature

- Syscall/API Tracer **Syscall**
- Syscall/Filesystem Monitor **Disk**
- TCPDump **Network**
- VMMemoryDump **Memory**

Cuckoo VM Support

- VirtualBox
- VMWare
- QEmu
- vsphere
- xen
- **bare metal**

Questions?

DEMO