Automatic Generation of Memory Analysis Plugins

Brendan Dolan-Gavitt
OMFW 2011

Memory Analysis Challenges

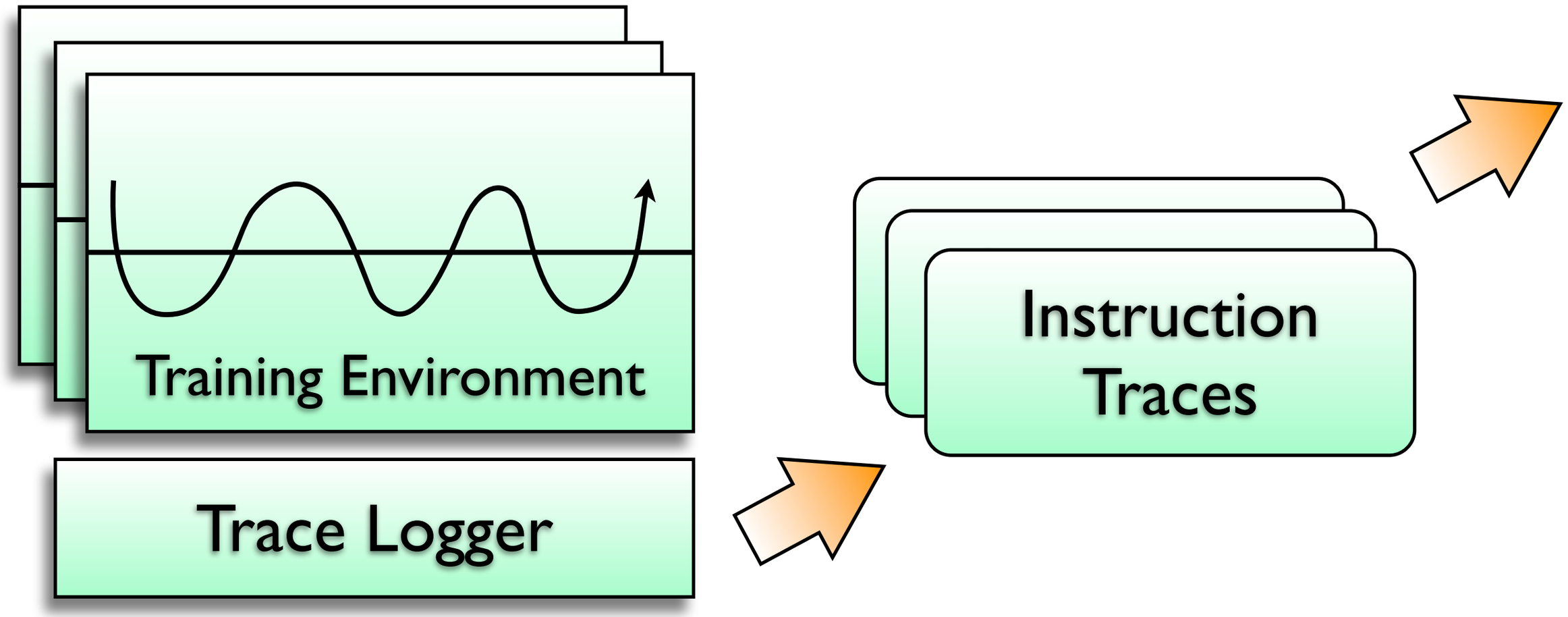
- Creating new plugins can take a lot of work
- Generally needs access to symbols or source code
- Reverse engineering required for closed source systems
- OS updates break our plugins!

Contrast: Live Analysis

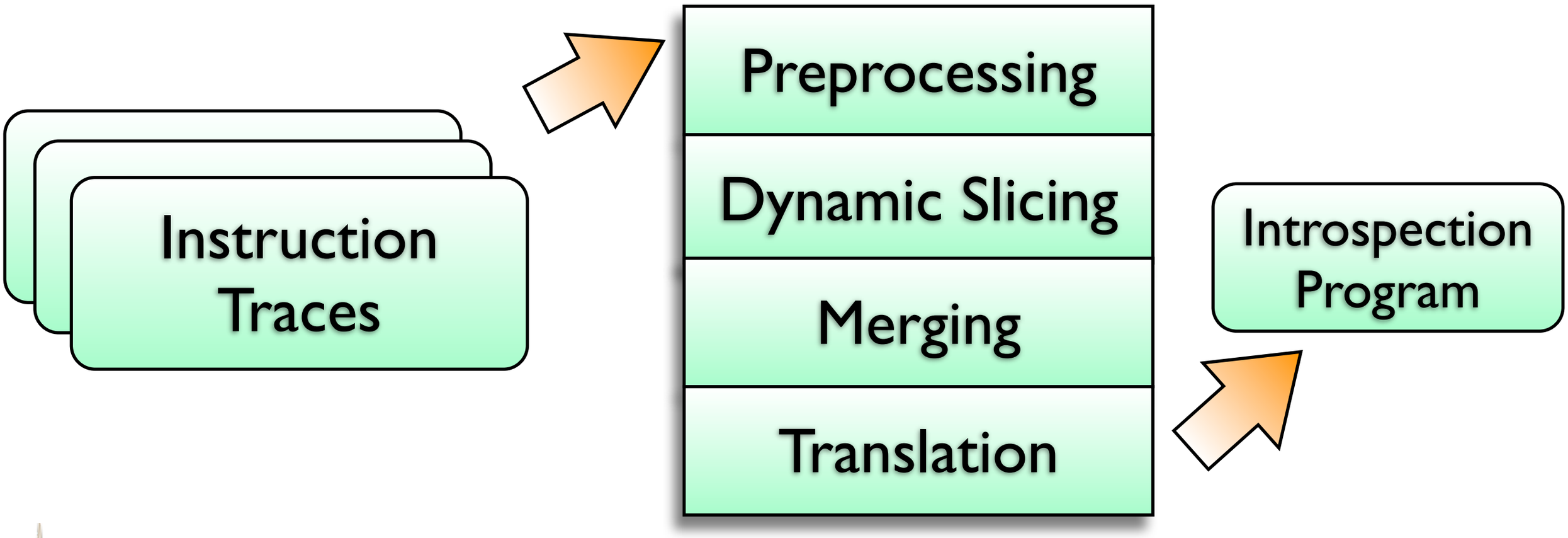
- Live analysis tools typically easier to write
- Can call existing APIs in the OS rather than reverse engineering
- But we lose the benefits of offline memory analysis
- Idea: can we convert live analysis tools into Volatility plugins?

Virtuoso

- Supports x86-based operating systems
- Runs live analysis "training program" and records all code executed
- Converts x86 code to Volatility plugin



Training Phase



Analysis Phase

Training

- Write *training program* using system APIs

```
#define __WIN32_LEAN_AND_MEAN__
#include <windows.h>
#include <psapi.h>
#pragma comment(lib, "psapi.lib")
#include <stdio.h>
#include "vmnotify.h"

int main(int argc, char **argv) {



  EnumProcesses(pids, 256, &outcb);

  return 0;
}
```

Training

- Write *training program* using system APIs

```
#define __WIN32_LEAN_AND_MEAN__
#include <windows.h>
#include <psapi.h>
#pragma comment(lib, "psapi.lib")
#include <stdio.h>
#include "vmnotify.h"

int main(int argc, char **argv) {
  DWORD *pids = (DWORD *) malloc(256);
  DWORD outcb;


  EnumProcesses(pids, 256, &outcb);

  return 0;
}
```

Training

- Annotate program with start/end markers

```
#define __WIN32_LEAN_AND_MEAN__
#include <windows.h>
#include <psapi.h>
#pragma comment(lib, "psapi.lib")
#include <stdio.h>
#include "vmnotify.h"

int main(int argc, char **argv) {
  DWORD *pids = (DWORD *) malloc(256);
  DWORD outcb;

  vm_mark_buf_in(&pids, 4);
  EnumProcesses(pids, 256, &outcb);
  vm_mark_buf_out(pids, 256);
  return 0;
}
```

Training

- Run program in QEMU to generate *instruction trace*
- Traces are in QEMU μOp format

```
INTERRUPT(0xfb,0x200a94,0x0)
TB_HEAD_EIP(0x80108028)
MOVL_T0_IM(0x0)
OPREG_TEMPL_MOVL_A0_R(0x4)
SUBL_A0_4()
OPS_MEM_STL_T0_A0(0x1,0xf186fe8,0x8103cfe8,
                  0xffffffff,0x215d810,0x920f0,0x0)
OPREG_TEMPL_MOVL_R_A0(0x4)
MOVL_T0_IM(0xfb)
OPREG_TEMPL_MOVL_A0_R(0x4)
SUBL_A0_4()
OPS_MEM_STL_T0_A0(0x1,0xf186fe4,0x8103cfe4,
                  0xffffffff,0x215d810,0x920f0,0xfb)
```

Trace Analysis

- What subset of this trace is relevant?
- System may have been doing other things in addition to just the operation we wanted
- Traces are processed to remove unwanted code:
  - Remove interrupts
  - Use program analysis (dynamic slicing) to determine exactly which instructions are necessary

Program Translation

- Goal: convert x86 → Volatility
- Changes:
  - Memory reads come from memory image
  - Memory writes are copy-on-write
  - CPU registers become program variables

Translation Example

Original x86

```
test byte [ebp+0x1c],0x10
mov edi,ebx
jnz 0xc02533a9
```

QEMU µOps

```
[TB @0xc0253368L *]
   IFLO_TB_HEAD_EIP(0xc0253368)
   IFLO_INSN_BYTES(0xc0253368,'f6451c10')
 * IFLO_OPREG_TEMPL_MOVL_A0_R(0x5)
 * IFLO_ADDL_A0_IM(0x1c)
 * IFLO_OPS_MEM_LDUB_T0_A0(...)
 * IFLO_MOVL_T1_IM(0x10)
 * IFLO_TESTL_T0_T1_CC()
   IFLO_INSN_BYTES(0xc025336c,'89df')
 * IFLO_OPREG_TEMPL_MOVL_T0_R(0x3)
 * IFLO_OPREG_TEMPL_MOVL_R_T0(0x7)
   IFLO_INSN_BYTES(0xc025336e,'7539')
 * IFLO_SET_CC_OP(0x16)
 * IFLO_OPS_TEMPLATE_JZ_SUB(0x0,0x1)
   IFLO_GOTO_TB1(0x60afcab8)
   IFLO_MOVL_EIP_IM(0xc0253370)
   IFLO_MOVL_T0_IM(0x60afcab9)
   IFLO_EXIT_TB()
```

Translation Example

QEMU µOps

```
[TB @0xc0253368L *]
   IFLO_TB_HEAD_EIP(0xc0253368)
   IFLO_INSN_BYTES(0xc0253368,'f6451c10')
 * IFLO_OPREG_TEMPL_MOVL_A0_R(0x5)
 * IFLO_ADDL_A0_IM(0x1c)
 * IFLO_OPS_MEM_LDUB_T0_A0(...)
 * IFLO_MOVL_T1_IM(0x10)
 * IFLO_TESTL_T0_T1_CC()
   IFLO_INSN_BYTES(0xc025336c,'89df')
 * IFLO_OPREG_TEMPL_MOVL_T0_R(0x3)
 * IFLO_OPREG_TEMPL_MOVL_R_T0(0x7)
   IFLO_INSN_BYTES(0xc025336e,'7539')
 * IFLO_SET_CC_OP(0x16)
 * IFLO_OPS_TEMPLATE_JZ_SUB(0x0,0x1)
   IFLO_GOTO_TB1(0x60afcab8)
   IFLO_MOVL_EIP_IM(0xc0253370)
   IFLO_MOVL_T0_IM(0x60afcab9)
   IFLO_EXIT_TB()
```

Python

```
A0 = EBP
A0 += UInt(0x1c)
T0 = ULInt8(mem.read(A0,1))
T1 = UInt(0x10)
CC_DST = T0 & T1
T0 = EBX
EDI = T0
CC_OP = 0x16
if (Byte(CC_DST) == 0):
    raise Goto(0xc0253370)
raise Goto(0xc02533a9)
```

Demo: Haiku Memory Analysis

- Haiku: open-source BeOS clone
- Let's create a process lister for it

Training Program

```c
#include <kernel/OS.h>
#include <stdio.h>
#include "vmnotify.h"

int32 real_list_procs(int32 *pids) {
    team_info info;
    int32 i = 0;
    int32 cookie = 0;
    while (get_next_team_info(&cookie, &info) == B_OK) {
        pids[i] = info.team;
        i++;
    }
    return i;
}

int32 list_procs(int32 *pids) {
    int i;
    vm_mark_buf_in(&pids, 4);
    i = real_list_procs(pids);
    vm_mark_buf_out(pids, 1024);
    return i;
}
```

Limitations

- Relies on old version of QEMU (0.9.1) – doesn't support many new OSes
- Execution must stay within one process while tracing
- More complex programs require multiple traces to cover multiple paths through prog
- Self-modifying code, synchronization not supported

Conclusions

- Can currently automate many simple kinds of memory analysis
- *Not* a full replacement for manually created plugins
- Provides a great shortcut for new OSes