Execution trace and memory analysis

Featuring Pin and Protobuf

Matthieu Tardy

EPITA 2017

July 17, 2015

Introduction

Motivation

- Discover PIN API
- Overview of performances and scalability
- Simplify reverse engineering

What is an execution trace?

- Log of memory accesses
- Read/Write
- Mappings changes (mmap/remap/unmap)

Why?

- Code optimization
- Reverse engineering
- Visualization

Optimization

- Benchmarks
- Global memory usage
- Structure accesses

Reverse engineering

Exemple

Statically linked binaries

- Include libraries
- Huge amount of code

Possible approaches

- Full reverse
- Code tracing
- Signatures - IDA FLIRT
- Memory access tracing

Methods

Linux

- ptrace
- Valgrind IR

Windows

- Debugging API

Both

- Pin
- Static analysis

Why Pin?

- Windows / Linux
- Faster

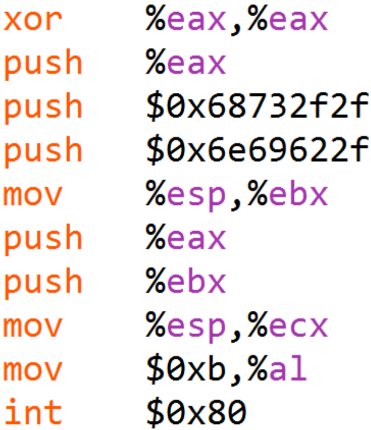
Pin

Instrumentation

What is it?

Inserting code in the program workflow.

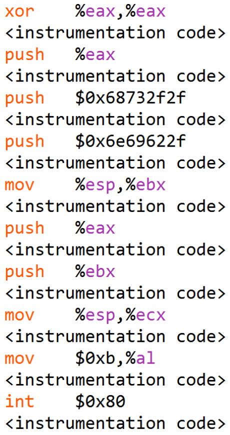
Instrumentation



```
xor     %eax,%eax
push    %eax
push    $0x68732f2f
push    $0x6e69622f
mov     %esp,%ebx
push    %eax
push    %ebx
mov     %esp,%ecx
mov     $0xb,%al
int     $0x80
```

Figure : Normal code

Instrumentation



```
xor     %eax,%eax
<instrumentation code>
push    %eax
<instrumentation code>
push    $0x68732f2f
<instrumentation code>
push    $0x6e69622f
<instrumentation code>
mov     %esp,%ebx
<instrumentation code>
push    %eax
<instrumentation code>
push    %ebx
<instrumentation code>
mov     %esp,%ecx
<instrumentation code>
mov     $0xb,%al
<instrumentation code>
int     $0x80
<instrumentation code>
```

Figure : Instrumented code

Features

- Instrumentation framework
- IA32 & x86_64
- JIT Mode (JIT recompiling)
- Probe mode (Code trampolines)
- Doesn't affect application state
- Handle dynamically generated code
- Attach to running process

How?

1. Generate a dynamic library (pintool)
2. Use pin to start / attach the process and inject the pintool
3. Process is now instrumented

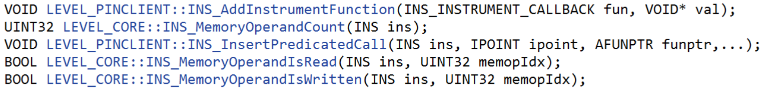
Core

Core

Goal

Record all the information we need at runtime to form the execution trace.

Memory access monitoring

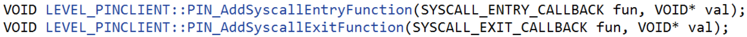


```
VOID LEVEL_PINCLIENT::INS_AddInstrumentFunction(INS_INSTRUMENT_CALLBACK fun, VOID* val);
UINT32 LEVEL_CORE::INS_MemoryOperandCount(INS ins);
VOID LEVEL_PINCLIENT::INS_InsertPredicatedCall(INS ins, IPOINT ipoint, AFUNPTR funptr,...);
BOOL LEVEL_CORE::INS_MemoryOperandIsRead(INS ins, UINT32 memopIdx);
BOOL LEVEL_CORE::INS_MemoryOperandIsWritten(INS ins, UINT32 memopIdx);
```

Guideline

1. Define an instruction callback function
2. Insert an instrumentation function if the instruction access to memory
3. Record the memory access

Memory mapping monitoring



```
VOID LEVEL_PINCLIENT::PIN_AddSyscallEntryFunction(SYSCALL_ENTRY_CALLBACK fun, VOID* val);
VOID LEVEL_PINCLIENT::PIN_AddSyscallExitFunction(SYSCALL_EXIT_CALLBACK fun, VOID* val);
```

Guideline

1. Define a syscall entry callback function
2. Record the syscall number
3. Define a syscall exit callback function
4. Check recorded syscall number and dump mappings if mmap/unmap/remap

Protobuf

What is protocol buffers?

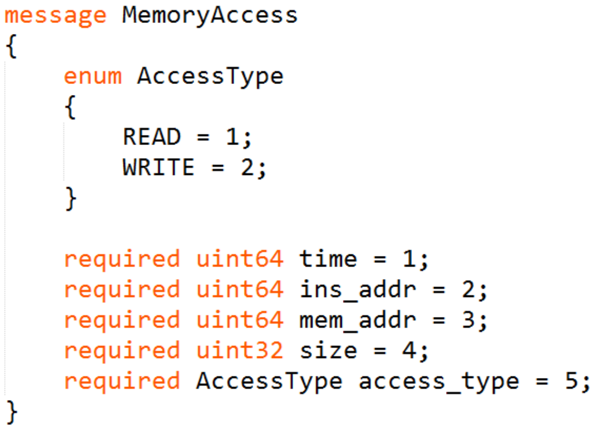
Method to serialize structured data

> *Think XML, but smaller, faster, and simpler.*
>
> *– Google*

Features

- Fast
- Flexible
- Easy to use
- Bindings for a lot of languages
- Adapted to large sets of small data

What does it look like?



```
message MemoryAccess
{
    enum AccessType
    {
        READ = 1;
        WRITE = 2;
    }

    required uint64 time = 1;
    required uint64 ins_addr = 2;
    required uint64 mem_addr = 3;
    required uint32 size = 4;
    required AccessType access_type = 5;
}
```

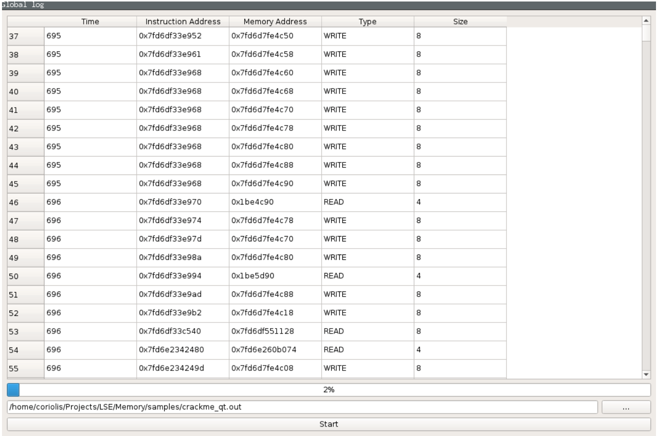
Analyzer

Analyzer

Goal

Analyze the informations in the execution trace and display it with a usable representation.

Basic log



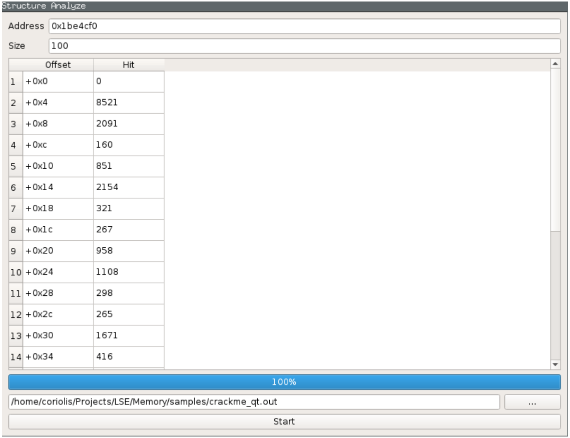
Global log

	Time	Instruction Address	Memory Address	Type	Size
37	695	0x7fd6df33e952	0x7fd6d7fe4c50	WRITE	8
38	695	0x7fd6df33e961	0x7fd6d7fe4c58	WRITE	8
39	695	0x7fd6df33e968	0x7fd6d7fe4c60	WRITE	8
40	695	0x7fd6df33e968	0x7fd6d7fe4c68	WRITE	8
41	695	0x7fd6df33e968	0x7fd6d7fe4c70	WRITE	8
42	695	0x7fd6df33e968	0x7fd6d7fe4c78	WRITE	8
43	695	0x7fd6df33e968	0x7fd6d7fe4c80	WRITE	8
44	695	0x7fd6df33e968	0x7fd6d7fe4c88	WRITE	8
45	695	0x7fd6df33e968	0x7fd6d7fe4c90	WRITE	8
46	696	0x7fd6df33e970	0x1be4c90	READ	4
47	696	0x7fd6df33e974	0x7fd6d7fe4c78	WRITE	8
48	696	0x7fd6df33e97d	0x7fd6d7fe4c70	WRITE	8
49	696	0x7fd6df33e98a	0x7fd6d7fe4c80	WRITE	8
50	696	0x7fd6df33e994	0x1be5d90	READ	4
51	696	0x7fd6df33e9ad	0x7fd6d7fe4c88	WRITE	8
52	696	0x7fd6df33e9b2	0x7fd6d7fe4c18	WRITE	8
53	696	0x7fd6df33c540	0x7fd6df551128	READ	8
54	696	0x7fd6e2342480	0x7fd6e260b074	READ	4
55	696	0x7fd6e234249d	0x7fd6d7fe4c08	WRITE	8

2%

/home/coriolis/Projects/LSE/Memory/samples/crackme_qt.out

Structure analysis



Address 0x1be4cf0

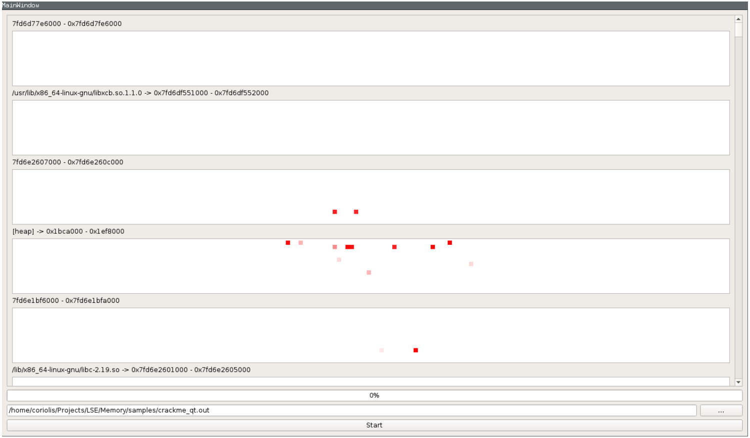
Size 100

	Offset	Hit
1	+0x0	0
2	+0x4	8521
3	+0x8	2091
4	+0xc	160
5	+0x10	851
6	+0x14	2154
7	+0x18	321
8	+0x1c	267
9	+0x20	958
10	+0x24	1108
11	+0x28	298
12	+0x2c	265
13	+0x30	1671
14	+0x34	416

100%

/home/coriolis/Projects/LSE/Memory/samples/crackme_qt.out

Dynamic Heatmap



MainWindow

7fd6d77e6000 - 0x7fd6d7fe6000

/usr/lib/x86_64-linux-gnu/libxcb.so.1.1.0 -> 0x7fd6df551000 - 0x7fd6df552000

7fd6e2607000 - 0x7fd6e260c000

[heap] -> 0x1bca000 - 0x1ef8000

7fd6e1bf6000 - 0x7fd6e1bfa000

/lib/x86_64-linux-gnu/libc-2.19.so -> 0x7fd6e2601000 - 0x7fd6e2605000

0%

/home/coriolis/Projects/LSE/Memory/samples/crackme_qt.out

Problems

- A *LOT* of data
  - Space problem
  - Speed problem
- Visualization isn't useful
  - Too much noise

What's the point then?

State

With this design it's not really usable.

But

- Logging using Pin is still the fastest way to have an execution trace of the *code*.
- Moving the analysis in the pintool allows us to do very efficient access analysis on structures or small areas.

What could be interesting

- DWARF Handling
- Code tracing

Conclusion

- Huge API, lot of features
- Not adapted for logging
- Viable for stats
- Viable for real instrumentation
  - Modify internal states registers
  - Modify functions

Tips

- Not compatible with 4.0 kernel
- Not compatible with gcc $>$ 4.9
- Be careful with multithreaded applications
- Use PIN Lock API
- Keep data processing in pintools

Outroduction

References

- Pin User Guide
- Pin presentation
- Protocol Buffers
- Malware unpacking with pin

Contact

- *Email*: coriolis@lse.epita.fr
- *Twitter*: @_coriolis_
- *IRC*: coriolis@Rezosup.org

Questions