Binary Code Analysis: Techniques, Tools, and Applications

Lecture 1: Introduction

Zhiqiang Lin

Department of Computer Science
University of Texas at Dallas

Outline

Acknowledgment: A portion of the slides in this lecture is compiled from presentations by Prof. Tom Reps and also Fish (author of `angr`)

Outline

What is Binary Analysis

The process of (automatically) reasoning/deriving properties about the structure/behavior/syntactics/semantics/anything of your interest of binary programs

```
zlin@zlin-desktop:~/$ hexdump -C /bin/ls|less
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  a4 45 40 00 00 00 00 00  |..>......E@.....|
00000020  40 00 00 00 00 00 00 00  70 96 01 00 00 00 00 00  |@.......p.......|
00000030  00 00 00 00 40 00 38 00  09 00 40 00 1c 00 1b 00  |....@.8...@.....|
00000040  06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000050  40 00 40 00 00 00 00 00  40 00 40 00 00 00 00 00  |@.@.....@.@.....|
00000060  f8 01 00 00 00 00 00 00  f8 01 00 00 00 00 00 00  |................|
00000070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00  |................|
00000080  38 02 00 00 00 00 00 00  38 02 40 00 00 00 00 00  |8.......8.@.....|
00000090  38 02 40 00 00 00 00 00  1c 00 00 00 00 00 00 00  |8.@.............|
...
```

Why Binary Code?

Access to the source code often is not possible:

- Proprietary software packages. (Volks Wagon's cheating software)
- Stripped executables.
- Proprietary libraries
- Malicious software (exploits), e.g., stuxnet

Why Binary Code?

Access to the source code often is not possible:

- Proprietary software packages. (Volks Wagon's cheating software)
- Stripped executables.
- Proprietary libraries
- Malicious software (exploits), e.g., stuxnet

Binary code is the only authoritative version of the program.

- Binary code is everywhere
- Changes occurring in the compile, optimize and link steps can create non-trivial semantic differences from the source and binary.
- What you see is not what you execute (WYSINWYX problem)

Why Binary Code?

- Windows
  - Login process keeps a user's password in the heap after a successful login
- To minimize data lifetime
  - clear buffer
  - call free()


```
memset(buffer, '\0', len);
free(buffer);
```

Why Binary Code?

- Windows
  - Login process keeps a user's password in the heap after a successful login
- To minimize data lifetime
  - clear buffer
  - call free()

- But . . .
  - the compiler might optimize away the buffer-clearing code ("useless-code" elimination)

```
memset(buffer, '\0', len);
free(buffer);
```

⇒

```
free(buffer);
```

Why Binary Code: Backdoor

Linux embedded device: HTTP server for management and video monitoring, with a known backdoor.

Backdoor!!!

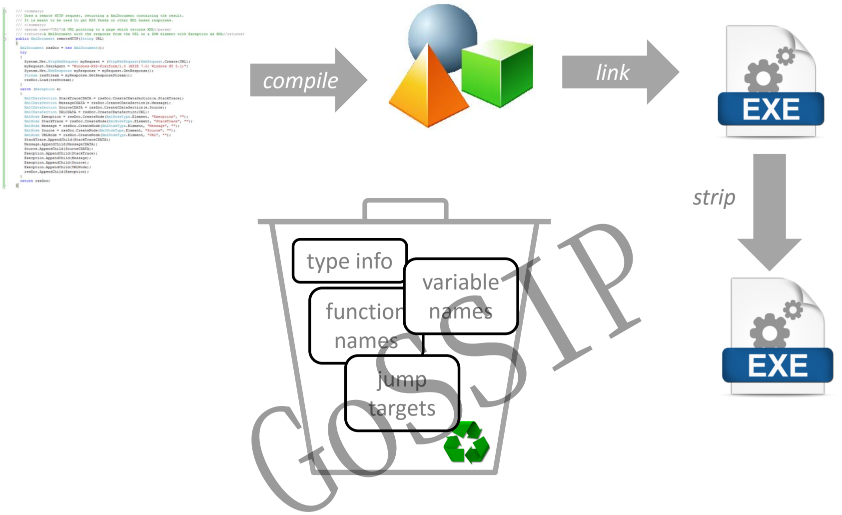
- ➔ Username: 3sadmin
- ➔ Password: 27988303

```
LDR    R1, =a3sadmin ; "3sadmin"
MOV    R0, R7 ; s1
BL     strcmp
CMP    R0, #0
LDR    R1, =a27988303 ; "27988303"
MOV    R0, R4 ; s1
```

Heffner, Craig. "Finding and Reversing Backdoors in Consumer Firmware." EELive! (2014).

WYSINWYX

What You See Is Not What You eXecute



An Example of WYSINWYX

```
int callee(int a, int b) {
  int local;
  if (local == 5) return 1;
  else return 2;
}

int main() {
  int c = 5;
  int d = 7;

  int v = callee(c,d);
  // What is the value of v here?
  return 0;
}
```

An Example of WYSINWYX

```
int callee(int a, int b) {
  int local;
  if (local == 5) return 1;
  else return 2;
}

int main() {
  int c = 5;
  int d = 7;

  int v = callee(c,d);
  // What is the value of v here?
  return 0;
}
```

Answer: 1
(for the Microsoft compiler)

Tutorial on x86 (Intel Syntax)

```
 p = q;
 p = *q;
*p = q;
 p = &a[2];
```

Tutorial on x86 (Intel Syntax)

```
 ecx = edx;
 ecx = *edx;
*ecx = edx;
 ecx = &a[2];
```

Tutorial on x86 (Intel Syntax)

```
mov     ecx, edx
```

```
 ecx = edx;
 ecx = *edx;
*ecx = edx;
 ecx = &a[2];
```

Tutorial on x86 (Intel Syntax)

```
mov     ecx,  edx
```

```
 ecx = edx;
 ecx = *edx;
*ecx = edx;
 ecx = &a[2];
```

Tutorial on x86 (Intel Syntax)

```
mov     ecx, edx
mov     ecx, [edx]
```

```
 ecx = edx;
 ecx = *edx;
*ecx = edx;
 ecx = &a[2];
```

Tutorial on x86 (Intel Syntax)

```
mov     ecx, edx
mov     ecx, [edx]
mov     [ecx], edx
```

```
 ecx = edx;
 ecx = *edx;
*ecx = edx;
 ecx = &a[2];
```

Tutorial on x86 (Intel Syntax)

```
mov     ecx, edx
mov     ecx, [edx]
mov     [ecx], edx
lea     ecx, [esp+8]
```

```
 ecx = edx;
 ecx = *edx;
*ecx = edx;
 ecx = &a[2];
```

Tutorial on x86 (Intel Syntax)

```
mov    ecx, edx
mov    ecx, [edx]
mov    [ecx], edx
lea    ecx, [esp+8]
```

```
 ecx = edx;
 ecx = *edx;
*ecx = edx;
 ecx = &a[2];
```

Stack pointer: **esp**

Frame pointer: **ebp**

An Example of WYSINWYX

```
int callee(int a, int b) {
  int local;
  if (local == 5) return 1;
  else return 2;
}

int main() {
  int c = 5;
  int d = 7;

  int v = callee(c,d);
  // What is the value of v here?
  return 0;
}
```

Answer: 1
(for the Microsoft compiler)

An Example of WYSINWYX

Standard prolog		Prolog for 1 local	
push	ebp	push	ebp
mov	ebp, esp	mov	ebp, esp
sub	esp, 4	push	ecx

```
int callee(int a, int b) {
  int local;
  if (local == 5) return 1;
  else return 2;
}

int main() {
  int c = 5;
  int d = 7;

  int v = callee(c,d);
  // What is the value of v here?
  return 0;
}
```

Answer: 1
(for the Microsoft compiler)

An Example of WYSINWYX

<u>Standard prolog</u>

```
push    ebp
mov     ebp, esp
sub     esp, 4
```

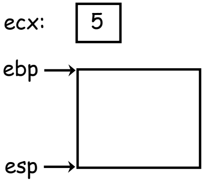
An Example of WYSINWYX

Standard prolog

push ebp

mov ebp, esp

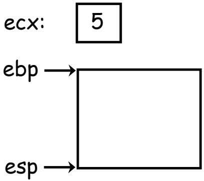
sub esp, 4



An Example of WYSINWYX

Standard prolog

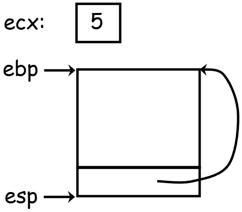
```
push    ebp
mov     ebp, esp
sub     esp, 4
```



An Example of WYSINWYX

<u>Standard prolog</u>

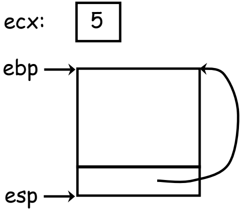
```
push    ebp
mov     ebp, esp
sub     esp, 4
```



An Example of WYSINWYX

<u>Standard prolog</u>

```
push    ebp
mov     ebp, esp
sub     esp, 4
```

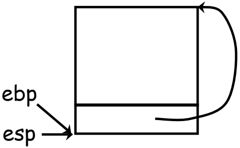


An Example of WYSINWYX

<u>Standard prolog</u>

```
push   ebp
mov    ebp, esp
sub    esp, 4
```

ecx: 5



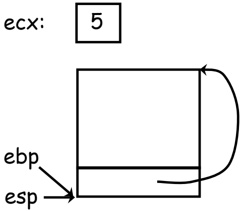
An Example of WYSINWYX

<u>Standard prolog</u>

push ebp

mov ebp, esp

$\boxed{\text{sub esp, 4}}$



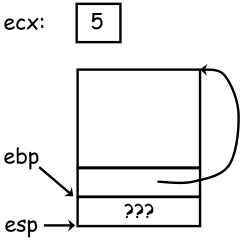
An Example of WYSINWYX

Standard prolog

push ebp

mov ebp, esp

sub esp, 4



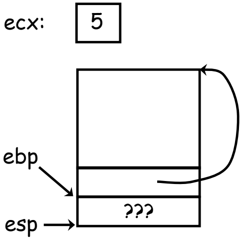
An Example of WYSINWYX

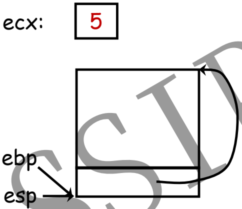
Standard prolog

```
push    ebp
mov     ebp, esp
sub     esp, 4
```

Prolog for 1 local

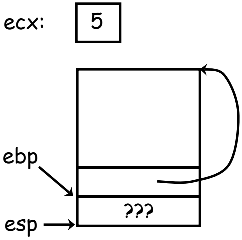
```
push    ebp
mov     ebp, esp
push    ecx
```

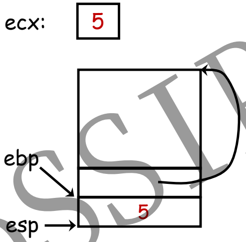




An Example of WYSINWYX

Standard prolog	Prolog for 1 local
push ebp	push ebp
mov ebp, esp	mov ebp, esp
sub esp, 4	push ecx





An Example of WYSINWYX

Standard prolog		Prolog for 1 local	
push	ebp	push	ebp
mov	ebp, esp	mov	ebp, esp
sub	esp, 4	push	ecx

```
int callee(int a, int
  int local;
  if (local == 5) return 1;
  else return 2;
}

int main() {
  int c = 5;
  int d = 7;

  int v = callee(c,d);
  // What is the value of v here?
  return 0;
}
```

Answer: 1
(for the Microsoft compiler)

An Example of WYSINWYX

```
int callee(int a, int b) {
  int local;
  if (local == 5) return 1;
  else return 2;
}
```

Standard prolog		Prolog for 1 local	
push	ebp	push	ebp
mov	ebp, esp	mov	ebp, esp
sub	esp, 4	push	ecx

Answer: 1
(for the Microsoft compiler)

```
int main() {
  int c = 5;
  int d = 7;

  int v = callee(c,d);
  // What is the value of v here?
  return 0;
}
```

```
mov   [ebp+var_8], 5
mov   [ebp+var_C], 7
mov   eax, [ebp+var_C]
push  eax
mov   ecx, [ebp+var_8]
push  ecx
call  _callee
. . .
```

An Example of WYSINWYX

Standard prolog		Prolog for 1 local	
push	ebp	push	ebp
mov	ebp, esp	mov	ebp, esp
sub	esp, 4	push	ecx

```
int callee(int a, int b) {
  int local;
  if (local == 5) return 1;
  else return 2;
}
```

Answer: 1
(for the Microsoft compiler)

```
int main() {
  int c = 5;
  int d = 7;

  int v = callee(c,d);
  // What is the value of v here?
  return 0;
}
```

```
mov   [ebp+var_8], 5
mov   [ebp+var_C], 7
mov   eax, [ebp+var_C]
push  eax
mov   ecx, [ebp+var_8]
push  ecx
call  _callee
. . .
```

Outline

What to Reason About in Binary Code?

The process of (automatically) reasoning/deriving properties about the structure/behavior/syntactics/semantics/anything of your interest of binary programs

1. What are the program's variables and their types?
2. What are the program's parameters and their types?
3. Where could this indirect jump go?
4. What function could be called at this indirect call site?
5. What could this dereference operation access/affect?
6. What kind of object is allocated at this allocation site?
7. What could the value held in V eventually affect?
8. What could have affected the value of V?
9. What are the statements (at instruction level) that contribute to the execution of i?
10. ...

Challenges: Abstraction Recovery

The first step in any binary code analysis is to reconstruct the abstractions distilled after compilation, such as recognizing the instructions, operand, opcode, variables, basic blocks, and control flows

Challenges: Abstraction Recovery

The first step in any binary code analysis is to reconstruct the abstractions distilled after compilation, such as recognizing the instructions, operand, opcode, variables, basic blocks, and control flows

Challenges

- Code/Data distinction
- Variable x86 instruction size
- Indirect Branches
- Functions without explicit CALL
- Position independent code (PIC)
- ...

It will be easier to recover these abstractions by using **dynamic analysis**, but will be much more challenge in **static analysis**.

Challenges: Path Coverage, and Path Explosion

For both static analysis and dynamic analysis, how to model the program execution path (too conservative, or too simple), and how to trigger the program path (especially for dynamic analysis) is another key challenge

Challenges: Path Coverage, and Path Explosion

For both static analysis and dynamic analysis, how to model the program execution path (too conservative, or too simple), and how to trigger the program path (especially for dynamic analysis) is another key challenge

Static analysis

- Too conservative
- Too many paths
- Impossible path

Dynamic analysis

- A single path
- Cover more path
- Test case generation

A Surprise:

Analyzing Executables can be Less Complicated than Analyzing Source

Many source-level issues gone

1. Use of multiple source languages
2. In-lined assembly code
3. Avoid building problems
4. Analyze the actual libraries
5. ...

Many people inspecting binaries in the whole life

1. IDA Pro Users
2. Anti-malware companies
3. Computer Emergency Response Teams
4. Malware writers
5. ...

A Surprise:

Executables can be a Better Platform for Finding Security Vulnerabilities

- Many exploits utilize platform-specific quirks
  - non-obvious and unexpected
  - compiler artifacts (choices made by the compiler)
    - Memory layout
      — padding between fields of a struct
      — which variables are adjacent
    - register usage
    - execution order
    - optimizations performed
    - compiler bugs

Outline

Basic Techniques

1. Data Flow Analysis
   - Data dependency
   - Taint analysis
   - Point-to analysis
2. Control Flow Analysis
   - Control flow graph
   - Call graph
   - Control dependency
3. Program Slicing
4. Symbolic Execution

Outline

Static Analysis, Dynamic Analysis, Symbolic Execution

BAP BAT CodeReason

radare2

vivisect Hex-Ray IDA rdis Valgrind

amoco fuzzgrind gdb

SemTrax

angr

BitBlaze

JARVIS

BARF

Jakstab

klee/s2e

insight

PIN QEMU Bindead

Triton

PySysEmu TEMU PEMU

miasm CodeSurfer

paimei

Common Tools

1. Static analysis
   - IDA Pro, BinNav
   - BAP
2. Dynamic analysis
   - PIN
   - QEMU
   - PEMU
3. Symbolic execution
   - FuzzBall, Fuzzgrind
   - S2E
   - Angr

Outline

Applications of Binary Analysis in Security

Use Cases

1. Reverse engineering (knowing the secrets)
2. Vulnerability discovery/fuzzing
3. Exploit generation
4. Software verification
5. Program testing
6. ...

Applications: Vulnerability Discovery

Vulnerability Discovery

How do I trigger path X or condition Y?

Basic Approaches

1. Static Analysis
   - “You can’t ” /“You might be able to”
   - Based on various static techniques.
2. Dynamic Analysis
   - Input A? Input B? Input C? ...
   - Based on concrete inputs to application
3. Symbolic Execution
   - Interpret the application.
   - Track “constraints” on variables.
   - When the required condition is triggered, “concretize” to obtain a possible input

Outline

Binary Analysis

- Binary code is everywhere, and it is the final representation of programs
- Binary analysis is challenging
- It is extremely useful to perform binary code analysis (vulnerability excavation, backdoor identification) in security
- Basic binary analysis approaches: static/dynamic analysis, symbolic execution
- There are many public available binary analysis tools

Binary Code Analysis: Techniques, Tools, and Applications

Lecture 2: Dynamic Binary Analysis

Zhiqiang Lin

Department of Computer Science
University of Texas at Dallas

Outline

Outline

What Is Instrumentation

A technique that inserts extra code into a program to collect information of your interest. Such technique has been widely used in practice in both program debugging and security analysis.

What Is Instrumentation

A technique that inserts extra code into a program to collect information of your interest. Such technique has been widely used in practice in both program debugging and security analysis.

```
Max = 0;
for (p = head; p; p = p->next)
{
  printf("In loop\n");
  if (p->value > max)
  {
    printf("True branch\n");
    max = p->value;
  }
}
```

What Is Instrumentation

A technique that inserts extra code into a program to collect information of your interest. Such technique has been widely used in practice in both program debugging and security analysis.

```
Max = 0;
for (p = head; p; p = p->next)
{
   count[0]++;
   if (p->value > max)
   {
      count[1]++;
      max = p->value;
   }
}
```

What Is (Dynamic) Binary Instrumentation

A technique that inserts extra code into the binary code of a program to collect (runtime) information of your interest.

What Is (Dynamic) Binary Instrumentation

A technique that inserts extra code into the binary code of a program to collect (runtime) information of your interest.

```
icount++
sub      $0xff, %edx
icount++
cmp      %esi, %edx
icount++
jle      <L1>
icount++
mov      $0x1, %edi
icount++
add      $0x10, %eax
```

What Can Instrumentation Do?

- Profiler for compiler optimization:
  - Basic-block count
  - Value profile
- Micro architectural study:
  - Instrument branches to simulate branch predictors
  - Generate traces
- Bug checking/Vulnerability identification/Exploit generation:
  - Find references to uninitialized, unallocated address
  - Inspect argument at particular function call
  - Inspect function pointers and return addresses
- Software tools that use dynamic binary instrumentation:
  - Valgrind, Pin, QEMU, DynInst, . . .

Instrumentation approaches: source vs. binary

- Source instrumentation:
  - Instrument source programs
- Binary instrumentation:
  - Instrument executables directly
- Advantages for binary instrumentation
  - Language independent
  - Machine-level view
  - Instrument legacy/proprietary software

Binary Instrumentation Is Dominant

- Libraries are a big pain for source code level instrumentation
  - Proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries).
- Easily handle multi-lingual programs
  - Source code level instrumentation is heavily language dependent.
    - More complicated semantics
- Turning off compiler optimizations can maintain an almost perfect mapping from instructions to source code lines
- Worms and viruses are rarely provided with source code
- ...

Instrumentation approaches: static vs. dynamic

- When to instrument
  - Instrument statically - before runtime
  - Instrument dynamically - during runtime
- Advantages for dynamic instrumentation
  - No need to recompile or relink
  - Discover code at runtime
  - Handle dynamically-generated code
  - Attach to running processes

How is Instrumentation used in Program Analysis?

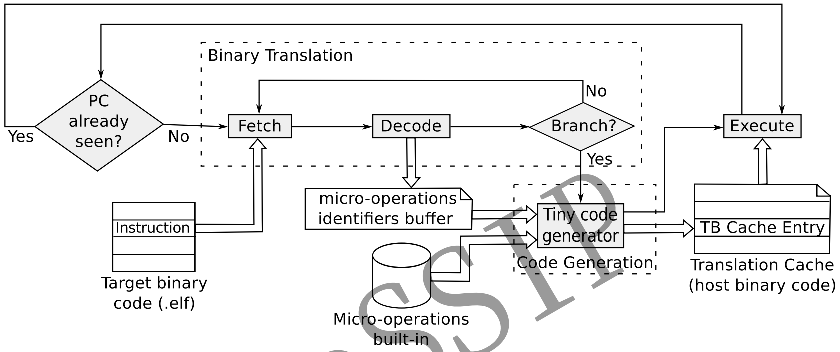
- Code coverage
- Call-graph generation
- Memory-leak detection
- Vulnerability identification
- Instruction profiling
- Data dependence profiling
- Thread analysis
  - Thread profiling
  - Race detection

Outline

QEMU

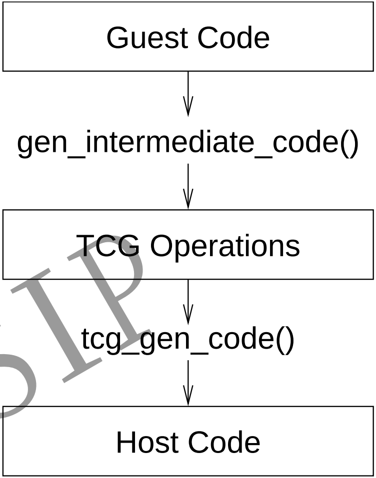
- QEMU is a generic and open source machine emulator and virtualizer.
- As a machine emulator, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). By using dynamic translation, it achieves very good performance.
- As a virtualizer, QEMU achieves near native performances by executing the guest code directly on the host CPU. QEMU supports virtualization when executing under the Xen hypervisor or using the KVM kernel module in Linux. When using KVM, QEMU can virtualize x86, server and embedded PowerPC, and S390 guests.

QEMU Internals

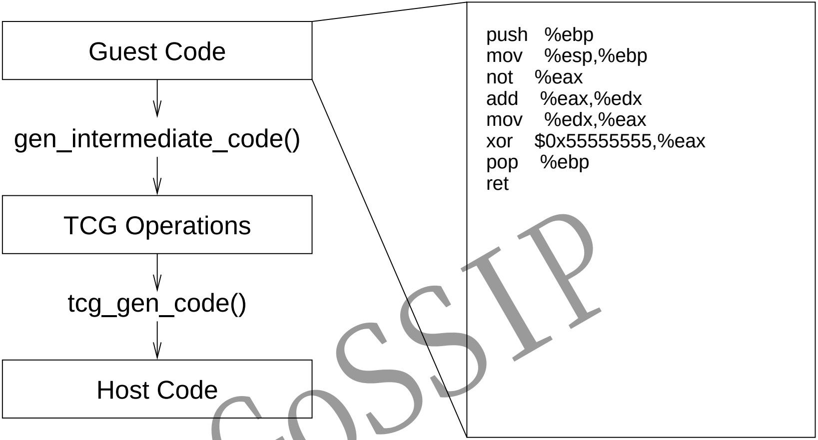


QEMU-Code Translation

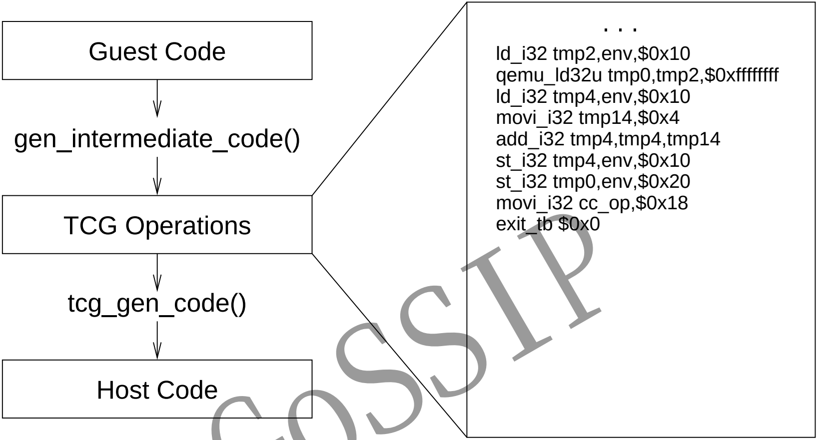
- QEMU uses an intermediate form.
- Frontends are in target-*/, includes alpha, arm, cris, i386, m68k,mips, ppc, sparc, etc.
- Backends are in tcg/*, includes arm/, hppa/, i386/, ia64/, mips/, ppc/, ppc64/, s390/, sparc/, tcg.c, tcg.h, tcg-opc.h, tcg-op.h, tcg-runtime.h



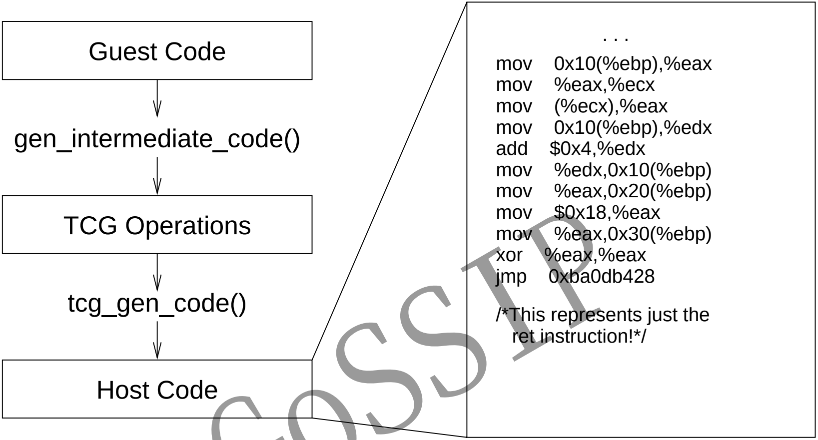
QEMU-Code Translation



QEMU-Code Translation



QEMU-Code Translation



QEMU-Code Base

- TranslationBlock structure in `translate-all.h`
- Translation cache is code gen buffer in `exec.c`
- `cpu-exec()` in cpu-exec.c orchestrates translation and block chaining.
- `target-*/translate.c`: guest ISA specific code.
- `tcg-*/*/`: host ISA specific code.
- `linux-user/*`: Linux usermode specific code.
- `vl.c`: Main loop for system emulation.
- `hw/*`: Hardware, including video, audio, and boards.

QEMU use cases

- Malware analysis
- Dynamic binary code instrumentation
- System wide taint analysis
- System wide data lifetime tracking
- Being part of KVM
- Execution replay
- ...

Outline

PIN

- Pin is a tool for the instrumentation of programs. It supports Linux* and Windows* executables for IA-32, Intel(R) 64, and IA-64 architectures.
- Pin allows a tool to insert arbitrary code (written in C or C++) in arbitrary places in the executable. The code is added dynamically while the executable is running. This also makes it possible to attach Pin to an already running process.

Credit: The rest of the slides are compiled from Intel's PIN tutorial

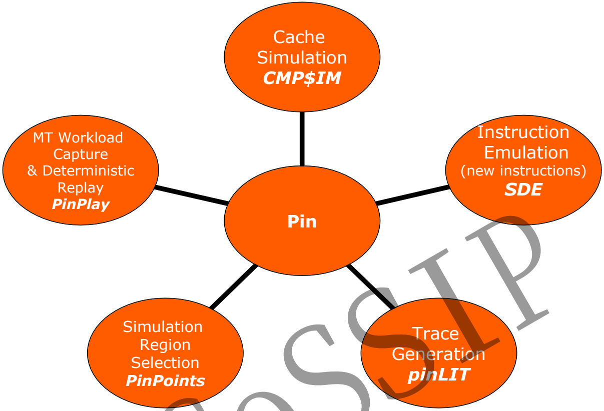
Advantages of Pin Instrumentation

- Easy-to-use Instrumentation:
  - Uses dynamic instrumentation
    - Does not need source code, recompilation, post-linking
- Programmable Instrumentation:
  - Provides rich APIs to write in C/C++ your own instrumentation tools (called Pintools)
- Multiplatform:
  - Supports x86, x86-64, Itanium
  - Supports Linux, Windows
- Robust:
  - Instruments real-life applications: Database, web browsers,. . .
  - Instruments multithreaded applications
  - Supports signals
- Efficient:
  - Applies compiler optimizations on instrumentation code

Pin Instrumentation Capabilities

1. Replace application functions with your own
    - Call the original application function from within your replacement function
2. Fully examine any application instruction, insert a call to your instrumenting function to be executed whenever that instruction executes
    - Pass parameters to your instrumenting function from a large set of supported parameters
        - Register values (including EIP), Register values by reference (for modification)
        - Memory address read/written by the instruction
        - Full register context
        - ...
3. Track function calls including syscalls and examine/change arguments
4. Track application threads
5. Intercept signals
6. Instrument a process tree
7. Many other capabilities ...

Example Pin-tools



Using Pin

- Launch and instrument an application:

$pin -t pintool.so - - application
1. instrumentation engine (provided)
2. instrumentation tool (write your own, or use a provided sample)

- Attach to a running process, and instrument application:

$pin -t pintool.so -pid 1234

Launch of the Instrumented Process

```
pin –t  pintool.dll -- application.exe
```

Launch of the Instrumented Process

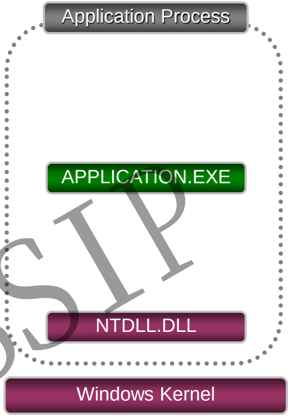
```
pin –t  pintool.dll -- application.exe
```

Launch of the Instrumented Process

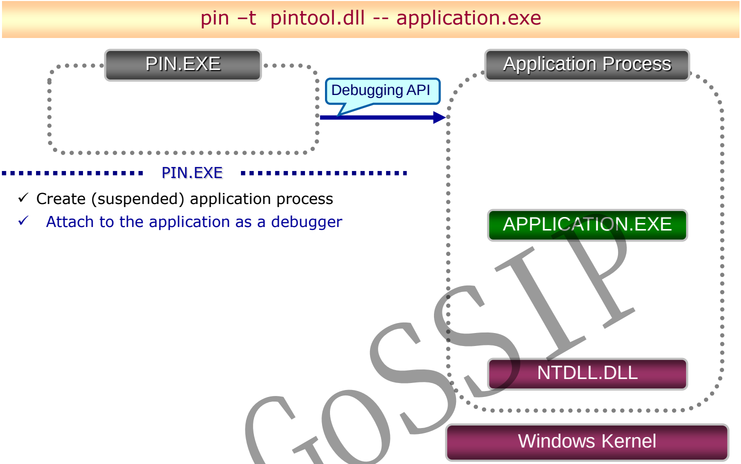
pin –t pintool.dll -- application.exe

PIN.EXE

✓ Create (suspended) application process

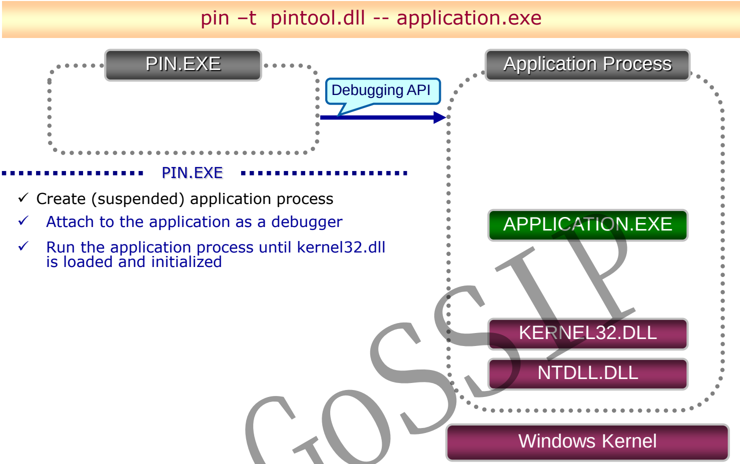


Launch of the Instrumented Process



- ✓ Create (suspended) application process
- ✓ Attach to the application as a debugger

Launch of the Instrumented Process

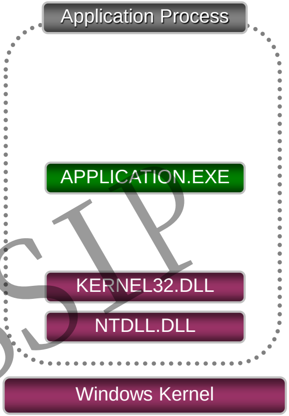


Launch of the Instrumented Process

pin –t pintool.dll -- application.exe

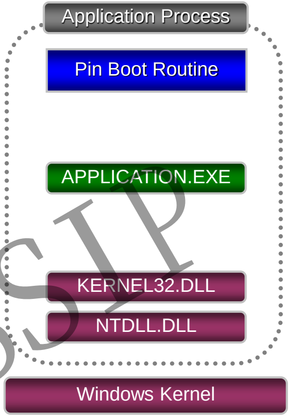
PIN.EXE

- ✓ Create (suspended) application process
- ✓ Attach to the application as a debugger
- ✓ Run the application process until kernel32.dll is loaded and initialized
- ✓ Detach from the application process



Launch of the Instrumented Process

pin –t pintool.dll -- application.exe

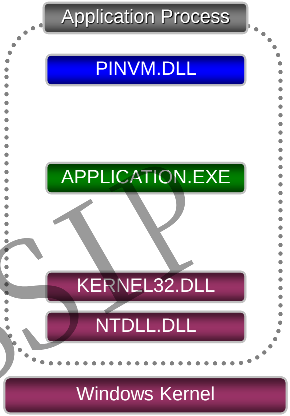


PIN.EXE

- ✓ Create (suspended) application process
- ✓ Attach to the application as a debugger
- ✓ Run the application process until kernel32.dll is loaded and initialized
- ✓ Detach from the application process
- ✓ Copy Boot Routine into the application process and set PC to start of the routine

Launch of the Instrumented Process

pin –t pintool.dll -- application.exe



PIN.EXE

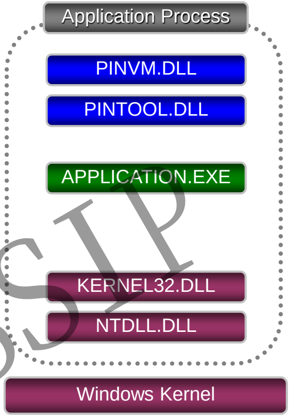
- ✓ Create (suspended) application process
- ✓ Attach to the application as a debugger
- ✓ Run the application process until kernel32.dll is loaded and initialized
- ✓ Detach from the application process
- ✓ Copy Boot Routine into the application process and set PC to start of the routine

Pin Boot Routine

- ✓ Load and start Pin VMM

Launch of the Instrumented Process

pin –t pintool.dll -- application.exe



PIN.EXE

- ✓ Create (suspended) application process
- ✓ Attach to the application as a debugger
- ✓ Run the application process until kernel32.dll is loaded and initialized
- ✓ Detach from the application process
- ✓ Copy Boot Routine into the application process and set PC to start of the routine

Pin Boot Routine

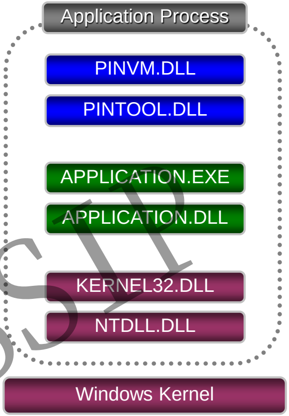
- ✓ Load and start Pin VMM

PINVM.DLL

- ✓ Load the instrumentation tool

Launch of the Instrumented Process

pin –t pintool.dll -- application.exe



PIN.EXE

- ✓ Create (suspended) application process
- ✓ Attach to the application as a debugger
- ✓ Run the application process until kernel32.dll is loaded and initialized
- ✓ Detach from the application process
- ✓ Copy Boot Routine into the application process and set PC to start of the routine

Pin Boot Routine

- ✓ Load and start Pin VMM

PINVM.DLL

- ✓ Load the instrumentation tool
- ✓ Instrument and execute the application

Launch of the Instrumented Process

pin –t pintool.dll -- application.exe

PIN.EXE

Application Process

PINVM.DLL

PINTOOL.DLL

APPLICATION.EXE

APPLICATION.DLL

KERNEL32.DLL

NTDLL.DLL

Windows Kernel

PIN.EXE

- ✓ Create (suspended) application process
- ✓ Attach to the application as a debugger
- ✓ Run the application process until kernel32.dll is loaded and initialized
- ✓ Detach from the application process
- ✓ Copy Boot Routine into the application process and set PC to start of the routine

Pin Boot Routine

- ✓ Load and start Pin VMM

PINVM.DLL

- ✓ Load the instrumentation tool
- ✓ Instrument and execute the application

All application instructions are executed under Pin control

Pin Instrumentation APIs

- Basic APIs are architecture independent:
  - Provide common functionalities like determining:
    - Control-flow changes
    - Memory accesses
- Architecture-specific APIs
  - e.g., Info about opcodes and operands
- Call-based APIs:
  - Instrumentation routines
  - Analysis routines

Instrumentation vs. Analysis

- Instrumentation routines define where instrumentation is inserted
  - e.g., before instruction
  - Occurs first time an instruction is executed
- Analysis routines define what to do when instrumentation is activated
  - e.g., increment counter
  - Occurs every time an instruction is executed

ManualExamples/itrace.cpp

```
#include <stdio.h>
%#include "pin.h"
FILE * trace;
void printip(void *ip) { fprintf(trace, "%p\n", ip); }
void Instruction(INS ins, void *v) {
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)printip, IARG_INST_PTR, IARG_END);
}
void Fini(INT32 code, void *v) { fclose(trace); }
int main(int argc, char * argv[]) {
    trace = fopen("itrace.out", "w");
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

Examples of Arguments to Analysis Routine

- IARG_INST_PTR
  - Instruction pointer (program counter) value
- IARG_UINT32 <value>
  - An integer value
- IARG_REG_VALUE <register name>
  - Value of the register specified
- IARG_BRANCH_TARGET_ADDR
  - Target address of the branch instrumented
- IARG_MEMORY_READ_EA
  - Effective address of a memory read
- And many more . . . (refer to the Pin manual for details)

Pintool Example: Instruction trace

- Need to pass the ip argument to the printip analysis routine

```
printip(ip)
sub          $0xff, %edx
printip(ip)
cmp          %esi, %edx
printip(ip)
jle          <L1>
printip(ip)
mov          $0x1, %edi
printip(ip)
add          $0x10, %eax
```

Running itrace tool

```
$ /opt/pin/pin
-t /opt/pin/source/tools/ManualExamples/
obj-intel64/itrace.so
-- /bin/ls

(...)
```

```
$ head itrace.out
0x7f907b188af0
0x7f907b188af3
0x7f907b189120
0x7f907b189121
0x7f907b189124
```

Reducing the Pintool's Overhead

Pintool's Overhead

Instrumentation Routines Overhead + Analysis Routines Overhead

Frequency of calling an Analysis Routine x Work required in the Analysis Routine

Work required for transiting to Analysis Routine + Work done inside Analysis Routine

Slower Instruction Counting

```
counter++;
sub   $0xff, %edx
counter++;
cmp   %esi, %edx
counter++;
jle   <L1>
counter++;
mov   $0x1, %edi
counter++;
add   $0x10, %eax
```

Faster Instruction Counting

Counting at BBL level

```
counter += 3
sub   $0xff, %edx
cmp   %esi, %edx
jle   <L1>
```

```
counter += 2
mov   $0x1, %edi
add   $0x10, %eax
```

Counting at Trace level

```
sub   $0xff, %edx
cmp   %esi, %edx
jle   <L1>
```

```
mov   $0x1, %edi
add   $0x10, %eax
counter += 5
```

```
counter+=3
```

L1

Writing your own Pintool

- It's easier to modify one of the existing tools, and re-use the existing makefile
- Install PIN package in your home directory, and work from there
  - /opt/pin-<version>-<architecture>-<os>.tar.gz

Outline

Summary

QEMU
open source processor emulator

Valgrind

Pin

Dyn
inst

GOSSIP

References

- http://en.wikipedia.org/wiki/QEMU
- http://wiki.qemu.org/Main_Page
- http://valgrind.org/
- http://www.pintool.org/
- http://www.dyninst.org/