CS451 – Software Analysis

Lecture 12

Binary Instrumentation

Elias Athanasopoulos
elathan@cs.ucy.ac.cy

What is binary instrumentation?

- Inserting new code to an existing binary is called *binary instrumentation*
  - This code can just observe or even modify the binary’s behavior
- The point where the new code is inserted is called *instrumentation point*
- The new code is called *instrumentation code*
- Examples
  - Profiling: add code for logging the time each function is executing
  - Security defense: modify all function epilogues to check the stack for integrity violation

Static vs dynamic

- Static Binary Instrumentation (SBI)
  - Use of binary re-writing to permanently modify binaries on the disk
- Dynamic Binary Instrumentation (DBI)
  - Insert instrumentation code while a process is executing
  - Debuggers do this for software breakpoints
  - A DBI engine may be much more generic and richer in functionality

DBI vs SBI

Dynamic Instrumentation	Static Instrumentation
- Relatively slow (4 times or more)	+ Relatively fast (10% to 2 times)
- Depends on DBI library and tool	+ Stand-alone binary
+ Transparently instruments libraries	- Must explicitly instrument libraries
+ Handles dynamically generated code	- Dynamically generated code unsupported
+ Can dynamically attach/detach	- Instruments entire execution
+ No need for disassembly	- Prone to disassembly errors
+ Transparent, no need to modify binary	- Error-prone binary re-writing
+ No symbols needed	- Symbols preferable to minimize errors

SBI’s main problem

- Code incorporates data accesses and code transfers that use relative addresses
- Adding new code to an existing block of code will shift all addresses
- Adapting all addresses in large binaries is practically impossible

Example

- Add code to a new section in the binary
  - E.g., modify the ELF binary to include a new section with name .text_instrumented

```
31 f6           xor esi,esi
41 83 c4 01     add r12d,0x1
b9 c1 8a 41 00  mov ecx,0x42
ba 01 00 00 00  mov edx,0x1
48 83 c5 08     add rbp,0x8
```

Original code

```
31 f6           xor esi,esi
41 83 c4 01     add r12d,0x1
b9 c1 8a 41 00  mov ecx,0x42
e9 de ad be ef  jmp instrum
48 83 c5 08     add rbp,0x8
```

Instrumented code

```
; pre-instrumentation
mov edx, 0x1 # instrum
; post-instrumentation
jmp instrum_site
```

Instrumentation code

The instruction `jmp` is 5 bytes, therefore the operands of `mov` will be affected.

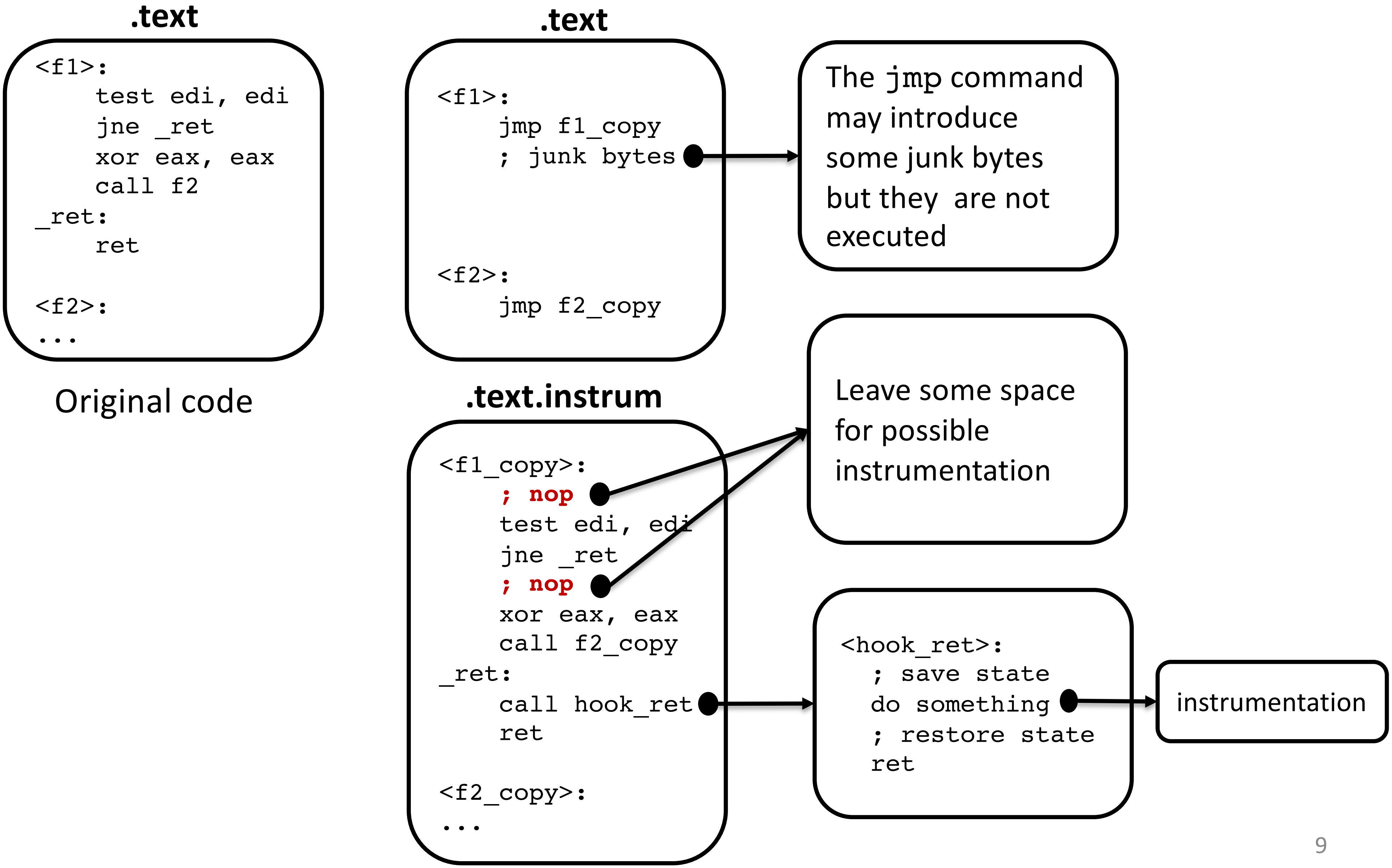
Using `int3`

- The main issue is that a jmp instruction, including the target at the instrumentation site is a multibyte injection
- Ideally, we want a 1-byte instruction to change the opcode at the instrumentation site
- `int3` is such a 1-byte an instruction
  - However, int3 needs the process to be traced using ptrace()

Trampoline

- Do not instrument the original code
- Copy the original code and instrument the new copy
- Use trampolines (jump instructions) to redirect the original code to the instrumented copy
- The binary does not break, since the copy of the code is instrumented
  - Control may be transferred to the original code, but a new trampoline will transfer control back to the instrumented code

Example



.text

```
<f1>:
    test edi, edi
    jne _ret
    xor eax, eax
    call f2
_ret:
    ret

<f2>:
...
```

Original code

.text

```
<f1>:
    jmp f1_copy
    ; junk bytes



<f2>:
    jmp f2_copy
```

The `jmp` command may introduce some junk bytes but they are not executed

.text.instrum

```
<f1_copy>:
    ; nop
    test edi, edi
    jne _ret
    ; nop
    xor eax, eax
    call f2_copy
_ret:
    call hook_ret
    ret

<f2_copy>:
...
```

Leave some space for possible instrumentation

```
<hook_ret>:
  ; save state
  do something
  ; restore state
  ret
```

instrumentation

Trampoline control flow

- When f1() is called, control will be transferred to f1_copy()
  - The `jmp` instruction may destroy some bytes in f1(), but this is not a problem anymore, since all code of f1() is copied to f1_copy()
- The SBI engine inserts several nop instructions in every possible instrumentation point of f1_copy()
  - Notice, there is some analysis here, for instance, the combo `test edi,edi; jne _ret` is preserved as is
- All direct jumps are replaced
  - `call f2` becomes call `f2_copy`

Handling indirect control flow

- All direct calls in the new section (.text.instrument) point to the copied functions
- Indirect calls cannot be changed, therefore they will point back to the original code
- When such indirect jumps are executed, control flow is transferred to the original code
  - But a trampoline will transfer the flow back to the instrumented code

Example of an indirect call

.text

```
<f1>:
    ...
     ; assume rax
     ; has f2
     call rax
<f2>:
    ...
```

Original code

.text

```
<f1>:
    jmp f1_copy
    ; junk bytes



<f2>:
    jmp f2_copy
```

.text.instrum

```
<f1_copy>:
    ...
    call rax
    ...

<f2_copy>:
    ...
```

Instrumented code

Indirect jumps

- A program may contain indirect jumps
  - E.g., C/C++ implements the switch statement with a jump table
- By default, the addresses of the jump table point to the original code
  - This is a code location in the *middle* of the function, where no trampoline code is near by
- Option 1: patch the jump table so that addresses point to the instrumented code
  - Dangerous, since valid data can be modified unintentionally
- Option 2: insert additional trampoline code in the middle of the function at every switch() case
  - Hard to find the exact location, which may in turn destroy code of other switch() cases

SBI reliability

- Indirect jumps, in the general case, may not be handled correctly
- Disassembly may have errors
- Some functions may be small to accommodate a 5-byte jump
- Inline data and code may cause some trampolines to overwrite valid data

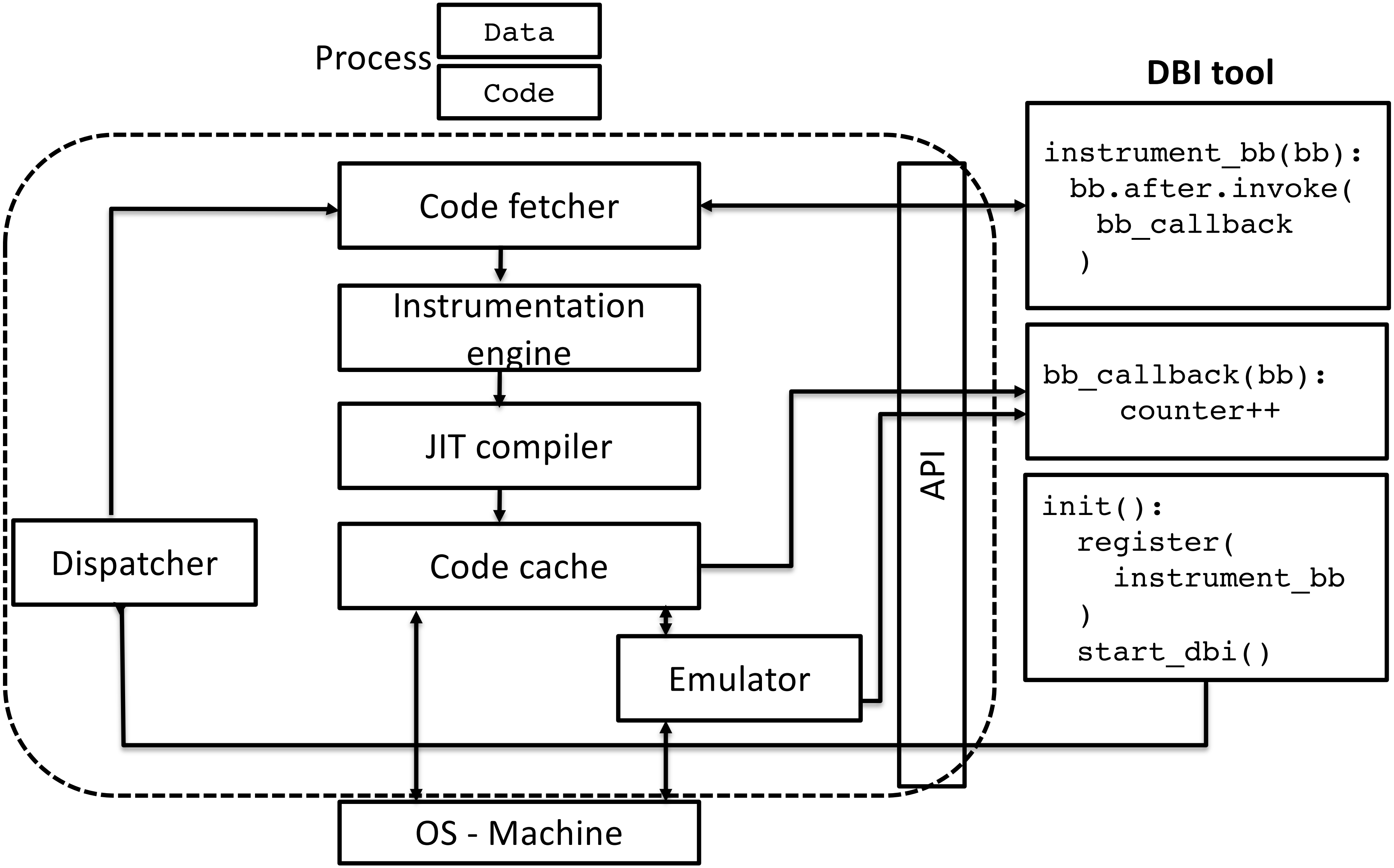
Dynamic binary instrumentation (DBI)

- DBI monitors the binary as it executes with the form of a process
- No need for accurate disassembly or for patching the existing code on disk
  - Less error-prone compared to SBI
  - Slower compared to SBI
- Available systems
  - Intel PIN, DynamoRIO

DBI internals

- The DBI engine is also based on debugging techniques
  - For instance, in Linux ptrace() will be used to monitor the executing engine
- Compared to a debugger, the DBI engine is much more complicated and richer in features to facilitate instrumentation
- The DBI engine exports an API for programmers to write their instrumentation code
  - The instrumentation code is compiled usually in a shared library
  - The API provides functions for handling various elements of the executing code (basic blocks, opcodes, etc.)

DBI Architecture



Process

Data

Code

DBI tool

```
instrument_bb(bb):
  bb.after.invoke(
    bb_callback
  )
```

```
bb_callback(bb):
     counter++
```

```
init():
  register(
    instrument_bb
  )
  start_dbi()
```

Remarks

- The programmer writes the DBI tool using the DBI API
  - Compiled as a shared library and loaded by the DBI engine
- The tool initialized the engine by registering a callback function for every basic block processed
  - The DBI engine will execute the callback whenever a conditional instruction is processed (end of a basic block)
- The DBI does not run the code directly, but fetches the code and instruments it before execution
- After instrumentation the code is optimized by the JIT compiler and is executed through the code cache

Executing the code

- The instrumented code lies in the code cache, and it is executed natively
- Some parts of the code may be emulated
  - For instance, system calls that can interfere with the instrumentation and process handling (such as `execve()`)
- The instrumented code contains additional code that is executed in parallel with the original code
  - For instance, a callback is executed at the end of each basic block
  - The DBI engine modifies each callback so that the state of the program is preserved

Homework

- Download PIN
  - https://www.cs.ucy.ac.cy/courses/EPL451/src/pin-3.6-97554-g31f0a167d-gcc-linux.tar.gz
  - Try to build and run a simple PIN tool