SKORPIO: Advanced Binary Instrumentation Framework

NGUYEN Anh Quynh <aquynh -at- gmail.com>

OPCDE Dubai, April 7th, 2018

About me

- Nguyen Anh Quynh, aquynh -at- gmail.com
  - Nanyang Technological University, Singapore
  - PhD in Computer Science
  - Operating System, Virtual Machine, Binary analysis, etc
  - Usenix, ACM, IEEE, LNCS, etc
  - Blackhat USA/EU/Asia, DEFCON, Recon, HackInTheBox, Syscan, etc
  - Capstone disassembler: http://capstone-engine.org
  - Unicorn emulator: http://unicorn-engine.org
  - Keystone assembler: http://keystone-engine.org

Agenda

1. Dynamic Binary Instrumentation (DBI)
2. Skorpio instrumentation engine
3. Demos
4. Conclusions

Dynamic Binary Instrumentation (DBI)

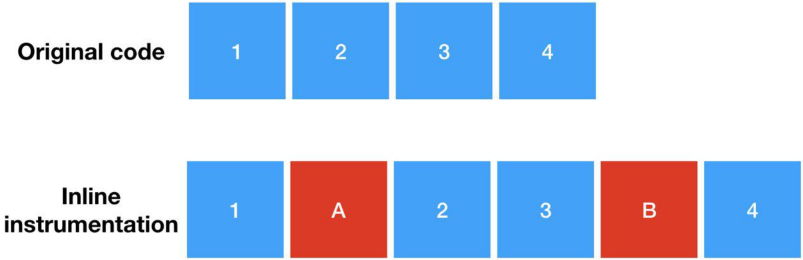
Definition

- A method of analyzing a binary application at runtime through injection of instrumentation code.
  - Extra code executed as a part of original instruction stream
  - No change to the original behavior
- Framework to build apps on top of it

Applications

- Code tracing/logging
- Debugging
- Profiling
- Security enhancement/mitigation

DBI illustration

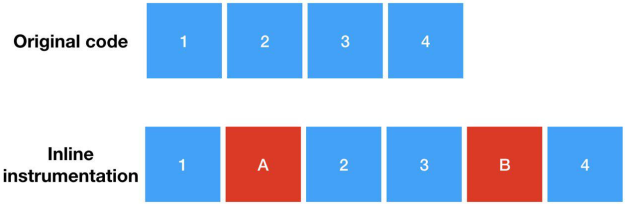


DBI techniques

- Just-in-Time translation
  - Transparently translate & execute code at runtime
    - Perform on IR: Valgrind
    - Perform directly on native code: DynamoRio
  - Better control on code executed
  - Heavy, super complicated in design & implementation
- Hooking
  - Lightweight, much simpler to design & implement
  - Less control on code executed & need to know in advance where to instrument

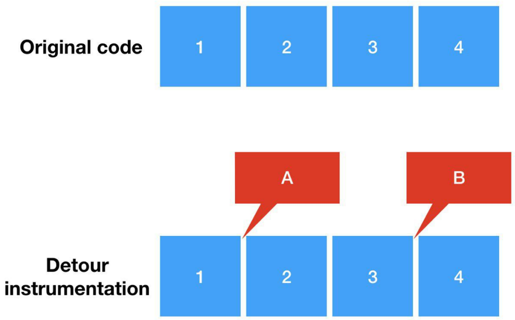
Hooking mechanisms - Inline

- Inline code injection
  - Put instrumented code inline with original code
  - Can instrument anywhere & unlimited in extra code injected
  - Require complicated code rewrite



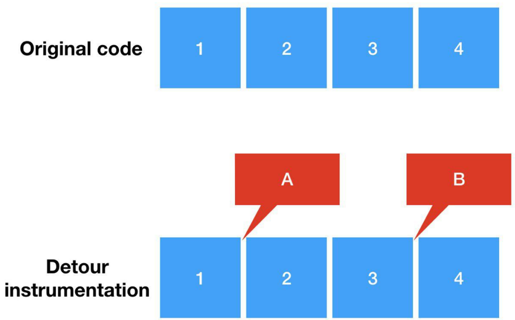
Hooking mechanisms - Detour

- Detour injection
  - Branch to external instrumentation code
    - User-defined **CALLBACK** as instrumented code
    - **TRAMPOLINE** memory as a step-stone buffer
  - Limited on where to hook
    - Basic block too small?
  - Easier to design & implement

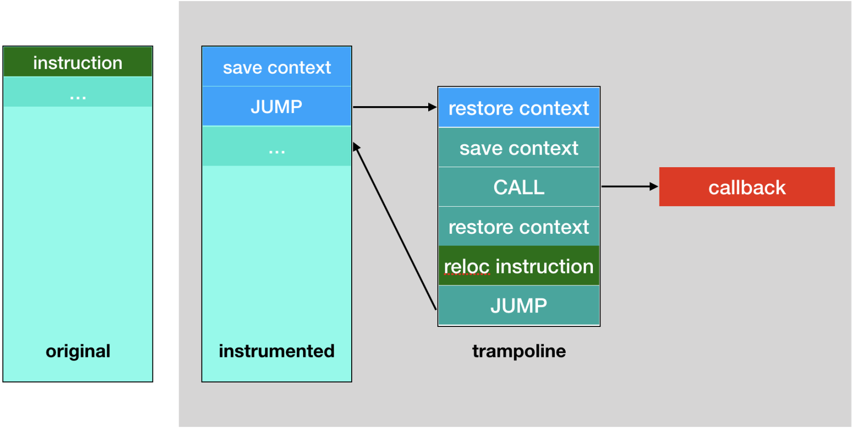


Detour injection mechanisms

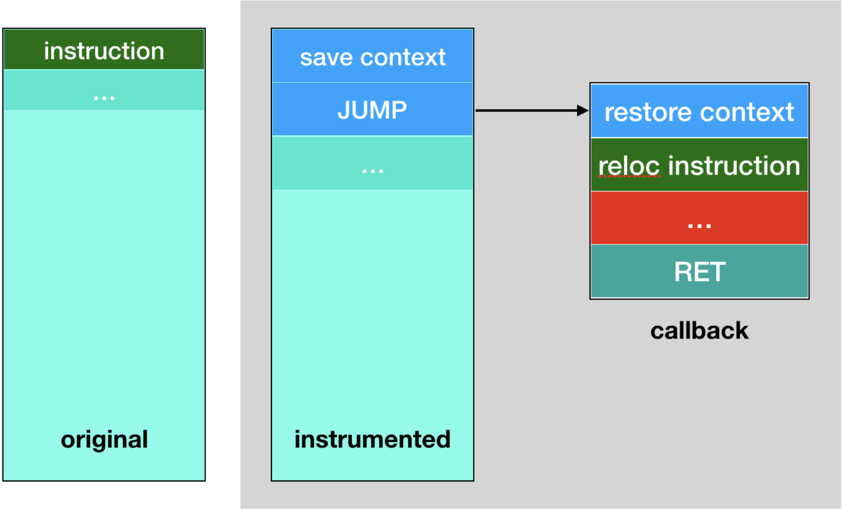
- Branch from original instruction to instrumented code
- Branch to trampoline, or directly to callback
  - Jump-trampoline technique
  - Jump-callback technique
  - Call-trampoline technique
  - Call-callback technique



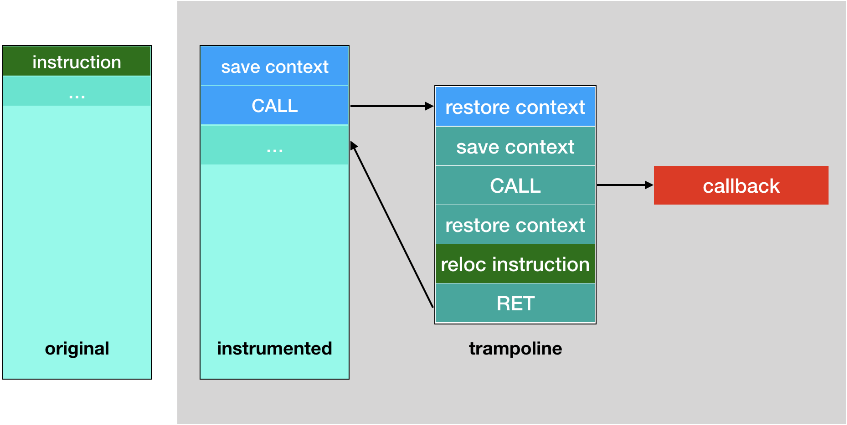
Jump-trampoline technique



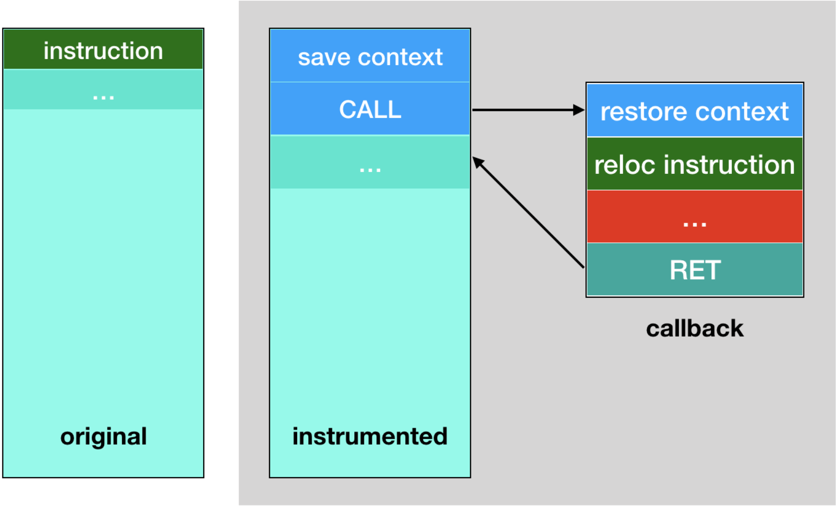
Jump-callback technique



Call-trampoline technique



Call-callback technique



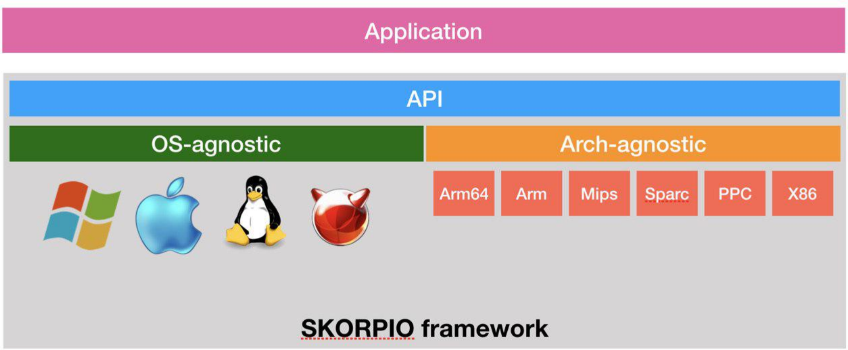
Problems of existing DBI

- Limited on platform support
- Limited on architecture support
- Limited on instrumentation techniques
- Limited on optimization

SKORPIO framework

- Open source, with permissive license
- Low level framework to build applications on top
  - App typically designed as dynamic libraries (DLL/SO/DYLIB)
- Cross-platform-architecture
  - Windows, MacOS, Linux, BSD, etc
  - X86, Arm, Arm64, Mips, Sparc, PowerPC
- Allow all kind of instrumentations
  - Arbitrary address, in any privilege level
- Designed to be easy to use, but support all kind of optimization
  - Super fast (100x) compared to other frameworks, with proper setup
- Support static instrumentation, too!

SKORPIO architecture



Cross platform - Memory

- Thin layer to abstract away platform details
- Different OS supported in separate plugin
  - Posix vs Windows
- Trampoline buffer
  - Allocate memory: malloc() vs VirtualAlloc()
  - Memory privilege RWX: mprotect() vs VirtualAlloc()
  - Trampoline buffer as close as possible to code to reduce branch distance
- Patch code in memory
  - Unprotect -> Patch -> Re-protect
  - mprotect() vs VirtualProtect()

Cross architecture - Save/Restore context

- Save memory/registers modified by initial branch & callback
- Keep the code size as small as possible
- Depend on architecture + mode
  - X86-32: PUSHAD; PUSHFD & POPFD; POPAD
  - X86-64 & other CPUs: no simple instruction to save all registers :-(
    - Calling convention: cdecl, optlink, pascal, stdcall, fastcall, safecall, thiscall, vectorcall, Borland, Watcom
    - SystemV ABI vs Windows ABI
- Special API to customize code to save/restore context

Cross architecture - Callback argument

- Pass user argument to user-defined callback
- Depend on architecture + mode & calling convention
  - SysV/Windows x86-32 vs x86-64
    - Windows: cdecl, optlink, pascal, stdcall, fastcall, safecall, thiscall, vectorcall, Borland, Watcom
  - X86-64: "mov rcx, <value>" or "mov rdi, <value>. Encoding depends on data value
  - Arm: "ldr r0, [pc, 0]; b .+8; <4-byte-value>"
  - Arm64: "movz x0, <lo16>; movk x0, <hi16>, lsl 16"
  - Mips: "li $a0, <value>"
  - PPC: "lis %r3, <hi16>; ori %r3, %r3, <lo16>"

Cross architecture - Branch distance

- Distance from hooking place to callback cause nightmare :-(
  - Some architectures have no explicit support for far branching
    - X86-64 JUMP: "push <addr>; ret" or "push 0; mov dword ptr [rsp+4], <addr>" or "jmp [rip]"
    - X86-64 CALL: "push <next-addr>; push <target>; ret"
    - Arm JUMP: "b <addr>" or "ldr pc, [pc, #-4]"
    - Arm CALL: "bl <addr>" or "add lr, pc, #4; ldr pc, [pc, #-4]"
    - Arm64 JUMP: "b <addr>" or "ldr x16, .+8; br x16"
    - Arm64 CALL: "bl <addr>" or "ldr x16, .+12; blr x16; b .+12"
    - Mips JUMP: "li $t0, <addr>; jr $t0"
    - Mips CALL: "li $t0, <addr>; move $t9, $t0; jalr $t0"
    - Sparc JUMP: "set <addr>, %l4; jmp %l4; nop"
    - Sparc CALL: "set <addr>, %l4; call %l4; nop"

Cross architecture - Branch for PPC

- PPC has no far jump instruction :-(
  - copy LR to r23, save target address to r24, then copy to LR for BLR
  - restore LR from r23 after jumping back from trampoline
  - "mflr %r23; lis %r24, <hi16>; ori %r24, %r24, <lo16>; mtlr %r24; blr"
- PPC has no far call instruction :-(
  - save r24 with target address, then copy r24 to LR
  - point r24 to instruction after BLR, so later BLR go back there from callback
  - "lis %r24, <target-hi16>; ori %r24, %r24, <target-lo16>; mtlr %r24; lis %r24, <ret-hi16>; ori %r24, %r24, <ret-lo16>; blr"

```
SK_INLINE_NO static void bbb_hook(size_t v)
{
    // restore LR from R24
    __asm__("mtlr %r24");

    printf("== in callback, userdata = %zu\n", v);

    return;
}
```

Cross architecture - Scratch register

- Scratch registers used in initial branching
  - Arm64, Mips, Sparc & PPC do not allow branch to indirect target in memory
  - Calculate branch target, or used as branch target
  - Need scratch register(s) that are unused in local context
    - Specified by user via API, or discovered automatically by engine

Cross architecture - Flush code cache

- Code patching need to be reflected in i-cache
- Depend on architecture
  - X86: no need
  - Arm, Arm64, Mips, PowrPC, Sparc: special syscalls/instructions to flush/invalidate i-cache
  - Linux/GCC has special function: cacheflush(begin, end)

Code boudary & relocation

- Need to extract instructions overwritten at instrumentation point
  - Determine instruction boundary for X86
  - Use Capstone disassembler
- Need to rewrite instructions to work at relocated place (trampoline)
  - Relative instructions (branch, memory access)
  - Use Capstone disassembler to detect instruction type
  - Use Keystone assembler to recompile

Code analysis

- Avoid overflow to next basic block
  - Analysis to detect if basic block is too small for patching
- Reduce number of registers saved before callback
- Registers to be choosen as scratch registers

Customize on instrumentation

- API to setup calling convention
- User-defined callback
- User-defined trampoline
- User-defined scratch registers
- User-defined save-restore context
- User-defined code to setup callback ars
- Patch hooks in batch, or individual
- User decide when to write/unwrite memory protect

Skorpio sample C code

```
Sample for Skorpio engine

--- Original code
BBB code = 0x400ca0, callback = 0x400c80

Hook info:
Hook type:                  2
Hook address:               0x400ca0
Hook callback:              0x400c80
Hook user_data:             0x7b
Hook trampoline addr:       0x7f1aa7911000
Hook trampoline size:       86
Hook trampoline code:       505351525756554150415141524154 9c48c7c77b0000006a00c70424321091a7c74424041a7f00006a00c70424800c4000c39d415c4
15a415941585d5e5f5a595b584883ec08b9800c4000baa00c400068ae0c4000c3
Patch size:                 14
Patched code:               ff2500000000001091a71a7f0000
Hook original code size:    14
Hook original code:         4883ec08b9800c4000baa00c4000


--- Functions with instrumentation now
== inside callback, userdata = 123
BBB code = 0x400ca0, callback = 0x400c80


--- Restored original code, now without instrumentation
BBB code = 0x400ca0, callback = 0x400c80
```

Status

- Cross-platforms: Windows, Linux, MacOS, BSD
- Python binding available
- Need to test on Android & iOS
- Cross-architecture: X86, Arm, Arm64, Mips, PowerPC, Sparc
- More test before public release - soon

Conclusions

- SKORPIO is an advanced framework for binary instrumentation
  - Open-source, cross-platform-architecture
  - All level of customization for better performance
  - Dynamic & static instrumentation
  - Lay the foundation for future security tools R&D

Acknowledgement

- Demo on Darko fuzzer is co-worked with Dr.Wei Lei (NTU)
- Huge thanks to @capstone_engine & @keystone_engine communities for great support!
- @_hugsy_ for Qemu images of Mips, PowerPC & Sparc

SKORPIO: Advanced Binary Instrumentation Framework

NGUYEN Anh Quynh <aquynh -at- gmail.com>