

# CSE509 : Computer System Security

# Binary Instrumentation

- R. Sekar
- Stony Brook University

# Limitations of SFI/NaCl Approach

- Need for compiler support
  - Does not work on arbitrary binaries --- binaries should have been compiled using a cooperative compiler
  - Otherwise, the binary will trivially fail the verification step
- Question: Can we instrument arbitrary COTS binaries to insert inline security checks?

# Motivation for COTS Binary Instrumentation

- ❑ No source code needed
  - Language-neutral (C, C++ or other)
- ❑ Can be largely independent of OS
- ❑ Ideally, would provide instruction-set independent abstractions
  - This ideal is far from today's reality
- ❑ Benefits
  - Application extension
    - Functionality
    - Security
    - Monitoring and debugging
  - Instrumenting long-running programs

# Approaches

- Static analysis/transformation
  - Binaries files are analyzed/transformed
  - Benefits
    - No runtime performance impact
    - No need for runtime infrastructure
  - Weakness
    - Error-prone, problem with signed code (can work around)

# Approaches

- Dynamic analysis/transformation
  - Code analyzed/transformed at runtime
  - Benefit: more robust/accurate
  - Weakness
    - High runtime overhead
    - Runtime complexity (infrastructure)

# Previous Works (Static)

- ❑ OM/ATOM (DEC WRL)
  - Proprietary and probably outdated
- ❑ EEL (Jim Larus et al, 1995)
  - The precursor of most modern rewriters
  - Targets RISC (SPARC)
  - Provides processor independent abstractions
  - Follow up works
    - UQBT (for RISC)
    - LEEL (for Linux/i386)

# Previous Works (Dynamic)

- ❑ LibVerify (Bell Labs/RST Corp)
  - Runtime rewriting for StackGuard
- ❑ DynamoRIO (HP Labs/MIT) Disassembles basic blocks at runtime
  - Provides API to hook into this process and transform executable
- ❑ Pin (Intel/U. Colorado)
- ❑ Valgrind
- ❑ A number of virtualization implementations rely binary translation (or used to)
- ❑ QEMU, VMWare, ...



# Phases in Static Analysis of Binaries

- ❑ Disassembly
- ❑ Instruction decoding/understanding
- ❑ Insertion of new code

# Static Disassembly

- Core component for static analysis of binaries
- Principal Approaches
  - Linear Sweep
  - Recursive Traversal

# Linear Sweep Algorithm

- ❑ Used by GNU objdump
- ❑ Problem:
  - There can be data embedded within code
    - There may also be padding, alignment bytes or junk
  - Linear sweep will incorrectly disassemble such data

```
Addr = startAddr;
while (Addr < endAddr )
{
    ins=decode(addr);
    addr+=LengthOf(ins);
}
```

# Linear Sweep Algorithm

804964c: 55 push %ebp	804964c: 55 push %ebp
804964d: 89 e5 mov %esp,%ebp	804964d: 89 e5 mov %esp,%ebp
804964f: 53 push %ebx	804964f: 53. push %ebx
8049650: 83 ec 04 sub \$0x4,%esp	8049650: 83 ec 04 sub \$0x4,%esp
8049653: eb 04 jmp 0x8049658	8049653: eb 04 jmp 0x8049658
8049655: e6 02 04 <junk>	8049655: e6 02 out 0x2, al
8049658: be 05000000 mov \$0x5,%esi	8049657: 04 be add al, 0xbe
	8049659: 05 00000012 add eax, 0x12000000

- Incorrectly disassembles junk (or padding) bytes
- Confusion typically cascades past the padding, causing subsequent instructions to be missed or misinterpreted.

# Self Repairing Disassembly

- ❑ Property of a disassembler where it re-synchronizes with the actual instruction stream
- ❑ Makes detecting disassembly errors difficult
  - 216 of 256 opcodes are valid
- ❑ Observation: re-synchronization happens quickly, within 2-3 instructions beyond point of error.

# Self Repairing Disassembly (example)

Consider the byte stream

55 89 e5 eb 03 90 90 83 0c 03 b8 01 00 00 00 c9

Linear Sweep output

```
100: push ebp
101: mov ebp, esp
103: jmp 109
105: nop
106: nop
107: or dword ptr ds:[ebx+eax*1], 0xb8
111: add dword ptr ds:[eax+eax*1], eax
113: add byte ptr ds:[eax+eax*1], al
116: leave
```

Correct Output

```
100: push ebp
101: mov ebp, esp
103: jmp 109
106: <GAP>
107: <GAP>
108: <GAP>
109: or al, 0x3
111: mov eax, 0x1
116: leave
```

# Recursive Traversal

- Approach: Takes into account the control flow behavior of the program
- Weakness: For indirect jumps, jump target cannot be determined statically, so no recursive traversal of the target can be initiated
- Some error cases not handled, e.g., jump to the middle of an instruction

```
RecursiveTraversal (addr) {  
    while (!visited[addr]) {  
        visited[addr] = true;  
        ins = decode (addr);  
        if (isControlTransfer(ins))  
            RecursiveTraversal (target(ins))  
        if (uncondJumpOrRet(ins))  
            return  
        else  
            addr+=LengthOf(ins);  
    }  
}
```

# Static Disassembly - Impediments

- ❑ Code/Data distinction
- ❑ Variable x86 instruction size
- ❑ Indirect Branches
  - Mainly due to function pointers
    - Cross-module calls
      - e.g., calls from executable to a library
    - GUI code (event handlers)
    - C++ code (Virtual functions)
  - Functions without explicit CALL
- ❑ PIC (Position-Independent Code)
- ❑ Hand-coded Assembly



# Optimized Code Example

```
#include <stdio.h>

void f(int c) {
    printf("%d\n", c);
}

void h(int i) {
    f(i+1);
}

int i(int j) {
    return j+1;
}

int main(int argc, char*argv[]) {
    h(i(argc));
    f(argc+2);
}
```

# Compiled Code Example

```
void f(int c) {  
    printf("%d\n", c);  
}
```

Function prologue

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp
```

```
pushl 8(%ebp)  
pushl $.LCO ("%d")  
call printf
```

Function epilogue

```
leave  
ret
```


# Optimized Code Example

```
void h(int i) {  
  f(i+1);  
}
```

h:

```
pushl %ebp  
movl %esp, %ebp  
subl $8, %esp
```

incl 8(%ebp)



No push of arguments

leave  
jmp f



No epilogue,  
not even a call!

# Optimized Code Example

```
main(int argc,  
     char*argv[]) {  
    h(i(argc));  
    f(argc+2);  
}
```

Return value in eax reg,  
No argument push!

No push of arguments  
to f, tail call

main:

```
    pushl %ebp  
    movl %esp, %ebp  
    pushl %ebx  
    subl $16, %esp  
    movl 8(%ebp), %ebx  
    pushl %ebx  
    call i  
    movl %eax, (%esp)  
    call h  
    addl $2, %ebx  
    movl %ebx, 8(%ebp)  
    addl $16, %esp  
    movl -4(%ebp), %ebx  
    leave  
    jmp f
```

# Static Code Transformation Limitations

- ❑ Cannot move code
  - Cannot predict the destination of indirect calls, so there is no safe way to move code
  - Can copy code, if original is left in place
    - If the goal is to protect against vulnerabilities in original code, then leaving of original code defeats this purpose!

# Static Code Transformation

## Limitations

- Code insertion is tricky
  - Obvious approach: overwrite original code with unconditional jump, patch
  - Problem: There may not be enough room for jump instruction (5 bytes)
  - Possible solution: use INT 3 (one-byte) instruction
    - Higher overhead for handling (signal generation/handling)

# Static Code Transformation Limitations

- ❑ Code insertion at arbitrary points is very difficult
  - Code insertion at beginning/end of function is easy
  - Other points in code are not well defined in optimized code
    - Loops may be unrolled
    - Switch statements translated to jump tables
    - Successive branches may be combined
    - Function arguments may not be explicitly pushed (nor return value popped)
    - Tail call optimization and function inlining

# Code Transformation

## Limitations

- ❑ Relocation of static data is not feasible
  - Cannot identify and relocate static pointers, which appear as immediate constants in assembly code
  - Note: These constants may be passed through several functions before used
- ❑ Note: Most above limitations can be removed if relocation information is present in binary
- ❑ Relocation of stack/heap data possible
  - Change SP at program beginning
  - Intercept/modify malloc and/or mmap requests



# Dynamic Transformation Techniques

# Libverify

- Inserts (StackShield) checks in binary code
  - Copy each function to heap
  - Modify first and last instruction in each function to jump to wrapper code implementing StackShield
  - Replace original copy with TRAP instructions
    - Any jump/calls to original code activates trap handler
    - Handler looks up corresponding address in copied code and transfers control there
    - If no copy exists (entry to function not discovered at load time), the function can be copied to heap and instrumented at this time

# Libverify

- Benefit: handles all indirect control transfers correctly
  - Note: indirect transfers will go to original code locations, unless code pointers are identified and modified to point to new locations
    - As mentioned before, this is hard to do in many cases, so this approach of using traps and runtime redirection is a good trade-off between performance and compatibility
- Drawback: performance impact if traps are repeatedly executed

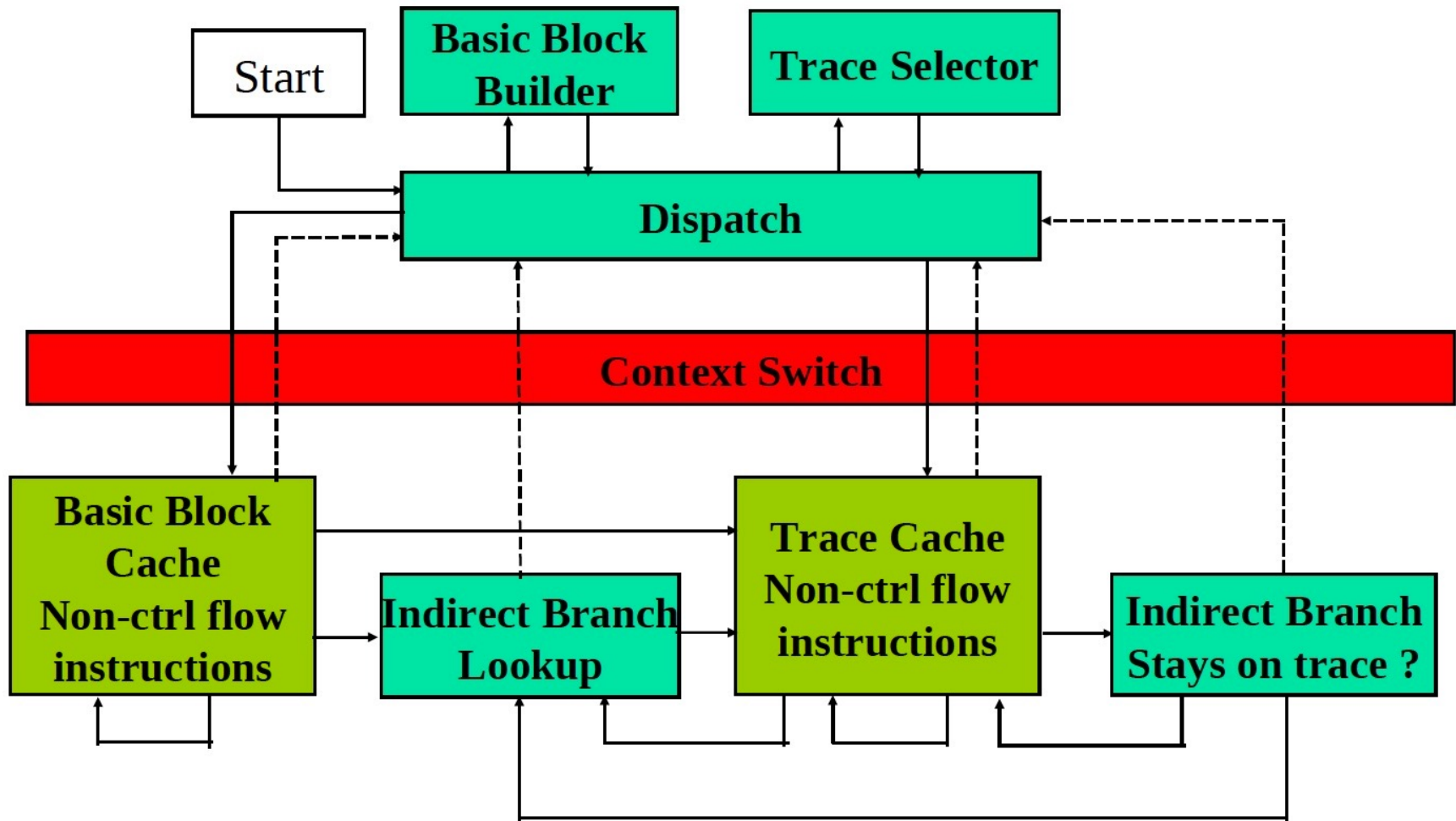
# DynamoRIO

- Use Libverify's runtime handling to the extreme
  - All code is discovered dynamically, analyzed dynamically, and then rewritten
  - Code is transformed one basic block at a time
    - Side-steps the thorny problem of disassembly
    - Note that it is trivial to reliably disassemble a single basic block, which is straight-line code with no control-transfers in the middle.
  - Only the first execution of a basis block requires analysis and rewriting. Subsequent executions can use the same rewritten block.

# DynamoRIO

- Control transfers occur in the last instruction of a basic block. These instructions need to be checked at runtime.
- Non-control-transfer instructions are executed natively

# RIO System Infrastructure



# DynamoRIO Operation

- Instrumented programs run in two contexts
  - DynamoRio context (above the redline, representing DynamoRIO runtime). Responsible for detecting the execution of new basic blocks (BBs)
    - These BBs are disassembled, analyzed and then transformed: just-in-time disassembly/rewriting, just before first execution
    - DynamoRIO provides an API for instrumentation: one can use this API to implement custom instrumentation, e.g., count number of BBs executed, number of memory accesses, etc.

# DynamoRIO Operation

- Instrumented programs run in two contexts
  - Application context (below the red line, application code executes natively)
    - Non-control-transfer instructions need no special treatment
    - Control-transfers need to be checked
      - If they are direct transfers, then we check if the target has already been instrumented (and hence is in the code cache). If so, directly jump there. If not, switch into DynamoRIO context to perform instrumentation.
      - Indirect transfers need to go through a translation table



# Handling Indirect CFT

- Note that indirect control transfers will use original code addresses
  - But the instrumented code is in the code cache at a different address. (We cannot use the original addresses, even if they were available: instrumentation causes code to expand, => every target except the very first in the instrumented application will reside at a different location as compared to the original code.)
  - As discussed, we cannot “fixup” code references either. Code addresses will be immediate constants in the binary, so no way to distinguish constants from code addresses
    - If we mistakenly “fixup” an integer value, program behavior changes
    - If we mistakenly omit the fixup of a code pointer, then code will jump to an incorrect location, likely leading to a crash

# Handling Indirect CFT

- Clever idea put forth by DynamoRIO authors
  - Wait until a pointer is actually used
    - If it is used as a target of control transfer, then it is obviously a code pointer
    - Just-in-time code pointer fixup: fixup happens at the very last step.

# Fixup Implementation

- Fixup is implemented using a translation table
  - A hash table `jmptab` maps the original address of a BB to its new address in the code cache (corresponding to the location of the instrumented version of code)
  - Each time DynamoRIO runtime instruments a BB, it enters the mapping between the original location and the new location in this table.

# Fixup Implementation

- ❑ At runtime, every indirect CFT to a location  $l$  is translated into `jmptab[l]`
  - Each indirect jump requires a hash table lookup, and has a performance cost
  - Fortunately, common cases (e.g., returns and repeated calls to same target) can be optimized
- ❑ If the target is not in `jmptab`, then control transferred to DynamoRIO runtime.

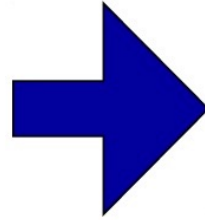
# DynamoRIO Context Switch

- Preserve the following conditions
  - All GPRs (8 in x86-32)
  - Eflags
  - Some system state. Eg: **error code**
    - DynamoRIO uses one slot in TLS (**thread local storage**) to store **error code (errno)** of the application.
      - DynamoRIO will use some library routines that may modify the state as error code, so it is necessary to preserve application's errno

# DynamoRIO Context Switch

```
add  %eax, %ecx
cmp  $4, %eax
jle  $0x40106f
```

bb with conditional jump



```
frag7: add  %eax, %ecx
       cmp  $4, %eax
       jle  <stub0>
       jmp  <stub1>
stub0: mov  %eax, eax-slot
       mov  &dstub0, %eax
       jmp  context_switch
stub1: mov  %eax, eax-slot
       mov  &dstub1, %eax
       jmp  context_switch
```

bb in code cache

- Assumes that the BBs at 0x40106f and the immediately following BBs are not in the code cache. In this case, control has to be transferred to DynamoRIO runtime when execution reaches the end of this BB. Before context switch, all of the application state (in particular, registers) need to be saved.

# Transparency & OS Issues

- ❑ Transparency: application cannot tell that it is running inside DynamoRIO
- ❑ Why does DynamoRIO need transparency?
  - Ensures that application behaves exactly the same way as before: it can't even tell the difference.
  - So, it can't evade DynamoRIO, nor can it behave differently.
- ❑ Transparency Issues
  - Library transparency
  - Thread transparency
  - Stack transparency
  - Address space transparency
  - Context Translation
  - Performance transparency  
(not preserved)

# Library Transparency

- ❑ **Issues** when both DynamoRIO and application enters the **same non-re-entrant library routine**
  - System state might be broken (errno)
  - Library routine may fail to work (malloc)
- ❑ **Solution:**
  - Use **system call** on both windows and Linux
  - Use **stateless library** routines
  - Implement own **memory (de)allocation routines.**



# Thread Transparency

- ❑ DynamoRIO does not create its own thread
- ❑ Why?
  - violate transparency when application that monitors all reads in a process
  - Performance issue when threads double
- ❑ What about one DynamoRIO thread?
  - Still violate transparency
  - Performance degrades when multiple threads switch into DynamoRIO mode
- ❑ **Therefore, use app thread with new context**

# Stack Transparency

- ❑ DynamoRIO does not “touch” application stack.
  - Some applications may access data **beyond the top of stack**. Eg: Microsoft office
  - Usual stack conventions may not be followed by hand-crafted assembly
    - use of esp as a GPR
  - Ability to read return address off stack and use in computing code location (or modify it)
    - Used in PIC (position-independent code)
- ❑ Solution:
  - **Use a private stack for each thread in DynamoRIO mode**
  - **Do not modify content of original stack**

# Address Space Transparency

- ❑ DynamoRIO should not “leak” information about itself.
  - On Windows, intercept
    - NtQueryVirtualMemory() that traverse memory regions
    - GetModuleFileName() (library call) to check if library is present
  - On Linux, intercept
    - mmap(). etc.
- ❑ More measures (security)
  - Mark DynamoRIO code as NX, when in code cache

# Context Translation

- When exception occurs, the faulting place should be **the original code address**.
  - Intercept user signal handler
  - Check the address map, find the **original address**
  - Modify the signal stack and go to user signal handler

# Transparency & OS Issues

- Operating System Issues
  - Kernel Mediated Control Flow
  - System Call Handling
  - Thread synchronization

# Kernel Mediated Control Flow

- Signal Handling
  - **DynamoRIO routine will get control first**
  - Signals will be queued and delayed, except urgent signals
    - Eg: SIGSEGV
- When signal arrives, if the thread is at
  - Code cache:
    - Unlink the current basic block, go back to DynamoRIO
    - If bb contains syscall, jump to exit stub before syscall.
      - **Why? Bound timing of signal handler, as syscall is expensive.**
  - DynamoRIO code:
    - Delay signal until reaching a safe place
    - Emulate kernel behavior

# System Call handling

- If syscall number is not statically known or on DynamoRIO's list
  - Insert pre-syscall & post-syscall routines around the instruction
- Uninterested syscall: **left unchanged. However:**
- For signal handling, app must LEAVE code cache QUICKLY (**for timing issue**)
  - Insert a jump prior to the syscall :
    - **Jmp <syscall or bail>**
    - **Bail: jmp <exit stub>**
    - **Syscall:**
    - **<system call instruction>**

# Program Shepherding: An IRM based on DynamoRIO

- Introduces in-line checks to defend against common exploits
  - Buffer overflow attacks
  - Format string attacks
  - Injection of malicious code
  - Re-use of existing code (existing code attacks)
- Sandboxing



# Program Shepherding Performance under Linux

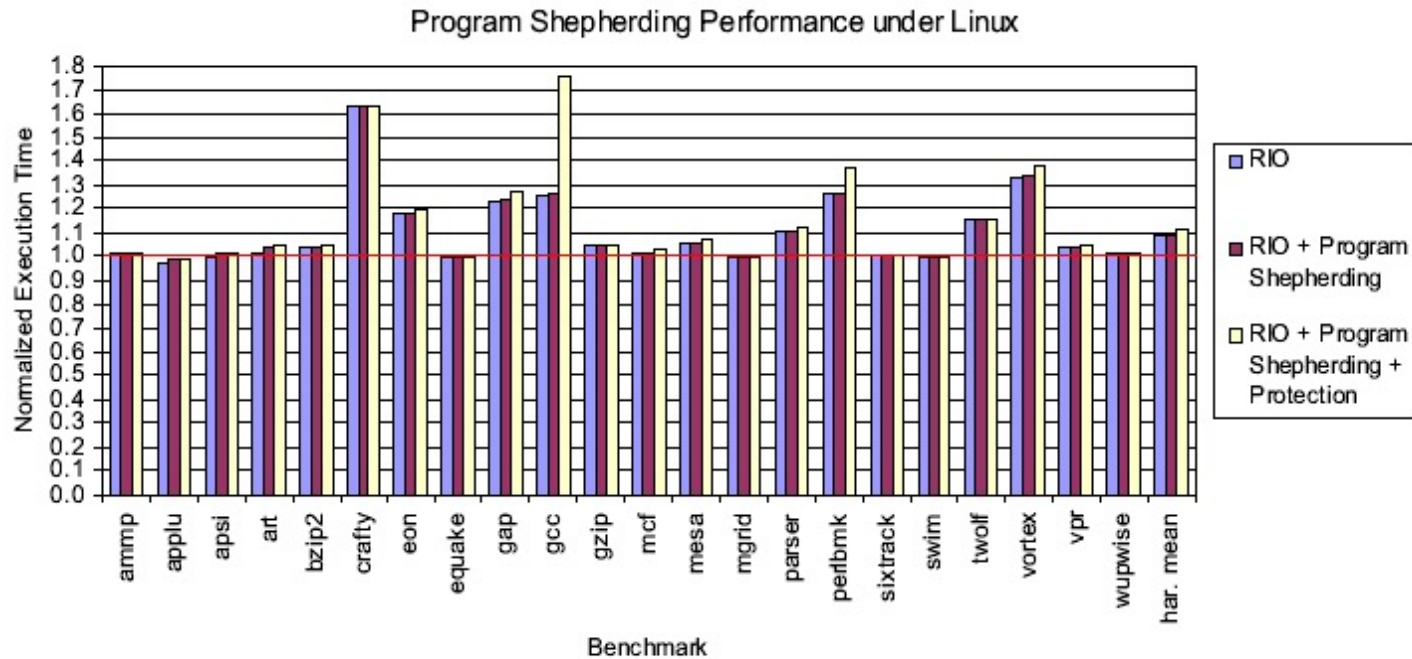


Figure 3: Normalized program execution time for our system (the ratio of our execution time to native execution time) on the SPEC2000 benchmarks [25] (excluding all FORTRAN 90 benchmarks) on Linux. They were compiled using `gcc -O3`. The final set of bars is the harmonic mean. The first bar is for RIO by itself; the middle bar shows the overhead of program shepherding (with the security policy of Table 1); and the final bar shows the overhead of the page protection calls to prevent attacks against the system itself.

- gcc is slow since it consists of many short runs with little code re-use

# Program Shepherding Performance under Windows

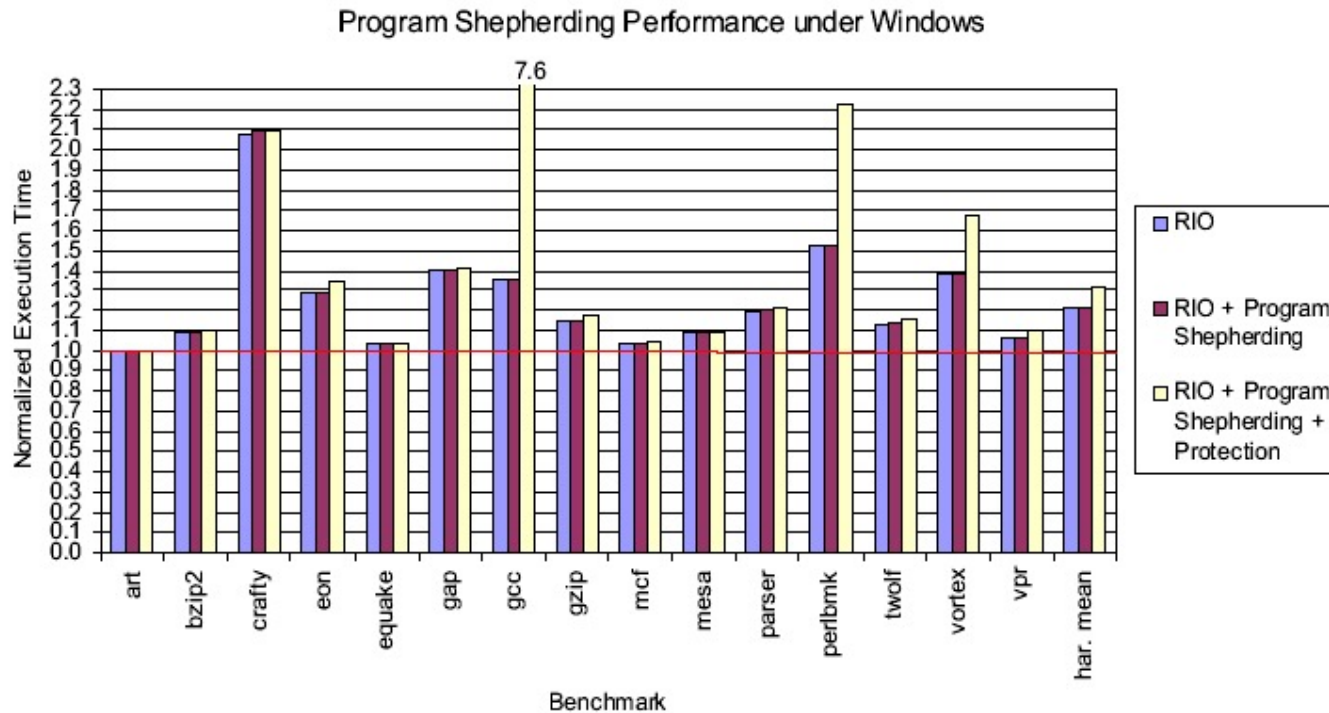


Figure 4: Normalized program execution time for our system (the ratio of our execution time to native execution time) on the SPEC2000 benchmarks [25] (excluding all FORTRAN benchmarks) on Windows 2000. They were compiled using `c1 /Ox`. The final set of bars is the harmonic mean. The first bar is for RIO by itself; the middle bar shows the overhead of program shepherding (with the security policy of Table 1); and the final bar shows the overhead of the page protection calls to prevent attacks against the system itself.

- ❑ Windows is much less efficient at changing privileges on memory pages than Linux

# Caveat about performance

- ❑ DBT performance measurements usually based very longrunning CPU-intensive benchmarks
- ❑ These applications represent the “best case scenario” for DBT systems
  - Rewrite once, execute for a long time
- ❑ Real-world performance can be bad
  - 10x to 40x slowdown in the worst case
- ❑ Example DBT systems
  - DynamoRIO, Pin, Valgrind, ...

# Caveat about performance

- But its exceptional level of compatibility with arbitrary binary code can still be compelling for
  - CPU-intensive applications with tight loops
  - Coarse-granularity instrumentation (i.e., very small fraction of instructions instrumented)
  - Debugging applications

# Other Dynamic Transformation Tools

## □ Pin

- better supported now than DynamoRIO
- better engineered for Linux

## □ Strata

## □ Valgrind

- Most popular open-source tool for finding memory errors and many other applications

## □ Qemu

- Can support whole system emulation

# DynamoRIO vs Pin

- ❑ Architecture dependency
  - Pin tools: written in c/c++
  - DynamoRIO: written in x86 assembly
- ❑ DynamoRIO's tools allow users to operate at a lower level
  - Have more control over efficiency, but programming can be hard, and architecture dependent.

# BBCount Pin Tool

- For more information, including tutorials and examples, see <https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>

```
static int bbcount;
VOID PIN_FAST_ANALYSIS_CALL docount() { bbcount++; }
VOID Trace(TRACE trace, VOID *v) {
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl);
         bbl = BBL_Next(bbl)) {
        BBL_InsertCall(bbl, IPOPOINT_ANYWHERE, AFUNPTR(docount),
                       IARG_FAST_ANALYSIS_CALL, IARG_END);
    }
}
int main(int argc, char *argv[]) {
    PIN_InitSymbols();
    PIN_Init(argc, argv);
    TRACE_AddInstrumentFunction(Trace, 0);
    PIN_StartProgram();
    return 0;
}
```

# BBCount DynamoRIO Tool

```
static int global_count;
static dr_emit_flags_t
event_basic_block(void *drcontext, void *tag, instrlist_t *bb,
                  bool for_trace, bool translating) {
    instr_t *instr, *first = instrlist_first(bb);
    uint flags;
    /* Our inc can go anywhere, so find a spot where flags are dead. */
    for (instr = first; instr != NULL; instr = instr_get_next(instr)) {
        flags = instr_get_arith_flags(instr);
        /* OP_inc doesn't write CF but not worth distinguishing */
        if (TESTALL(EFLAGS_WRITE_6, flags) && !TESTANY(EFLAGS_READ_6, flags))
            break;
    }
    if (instr == NULL)
        dr_save_arith_flags(drcontext, bb, first, SPILL_SLOT_1);
    instrlist_meta_preinsert(bb, (instr == NULL) ? first : instr,
                             INSTR_CREATE_inc(drcontext, OPND_CREATE_ABSMEM((byte *)&global_count, OPSZ_4)));
    if (instr == NULL)
        dr_restore_arith_flags(drcontext, bb, first, SPILL_SLOT_1);
    return DR_EMIT_DEFAULT;
}
DR_EXPORT void dr_init(client_id_t id) {
    dr_register_bb_event(event_basic_block);
}
```



# Applicability of Static Vs Dynamic Techniques

- ❑ Some techniques require static instrumentation
  - Any technique that uses static analysis to compute a property and then enforces it at runtime
    - CFI, some aspects of bounds-checking, some types of randomizations, ...
- ❑ Others can use dynamic instrumentation
  - Stackguard, SFI (but may be limited if CFI can't be assured)
- ❑ And yet others that cannot use static instrumentation
  - Obfuscated code, mainly malware

# Obfuscation against Disassembly

- ❑ Conditional jumps where the condition is always true (or false)
  - Use an opaque predicate to hide this
- ❑ Instructions that fault
  - Execution continues in exception handler
- ❑ Embedding data in the midst of code
  - With indirect jumps that make it impossible to distinguish between code and data

# Control-flow Obfuscation Against Reverse Engineering

- ❑ Split or aggregate
  - Basic blocks
  - Loops
    - e.g., one loop becomes two loops or vice-versa
  - Procedures
  - Replace one procedure by two or merge two procedures
  - Inline a procedure, or outline (i.e., create new procedure)
- ❑ Reorder

# Control-flow Obfuscation Against Reverse Engineering

- ❑ Insert dead-code (i.e., unreachable code)
  - Obfuscate using conditions
- ❑ Replace instruction sequences w/ alternate ones
- ❑ Insert conditional jumps using "opaque" predicates
- ❑ Insert indirect jumps
- ❑ Exploit aliasing and memory errors

# Data Obfuscation

- ❑ Rename variables
- ❑ Split or aggregate variables
  - Split structures into individual variables or vice-versa
- ❑ Split individual variables
  - E.g.,  $A = B - C$  - instead of  $A$ , use  $B$  and  $C$
  - Clone a variable
- ❑ Pad arrays (and possibly structures) with junk elements
- ❑ "Encrypt" data values

# Data Obfuscation

- ❑ Introduce extra levels of indirection
- ❑ Instead of a simple variable, declare a pointer
- ❑ Introduce aliasing
- ❑ Introduce memory errors
- ❑ Introduce additional (or remove) function parameters

# Questions