Binary Instrumentation

(and Reverse Engineering)

Fall 2024

R. Sekar

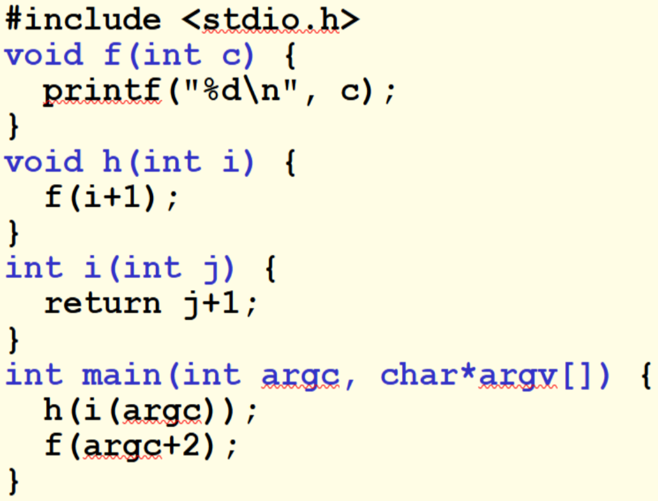
Limitations of NaCl/Wasm Approach

- 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

Challenges: Disconnect between source and binary code



```
#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);
}
```

Example: A C-function and its compiled code

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

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

pushl    8(%ebp)
pushl    $.LC0 ("%d")
call     printf

leave
ret
```

Function prologue

Function epilogue

Challenges in Optimized Code I

```
void h(int i) {  h:
  f(i+1);           pushl   %ebp
}                   movl    %esp, %ebp
                    subl    $8, %esp
```

No push of arguments

```
                    incl    8(%ebp)
```

No epilogue,
not even a call!

```
                    leave
                    jmp     f
```

Challenges in Optimized Code II

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

```
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
```

Return value in eax reg,
No argument push!

No push of arguments
  to f, tail call

Binary Instrumentation Approaches

- Static analysis/transformation
  - Strengths
    - No runtime performance impact
    - No need for runtime infrastructure
  - Weakness
    - Implementation challenges due to disassembly difficulty
- Dynamic analysis/transformation
  - Benefit: Side-steps disassembly difficulties
  - Weakness
    - High runtime overhead
    - Runtime complexity (infrastructure)

Phases in Static Analysis of Binaries

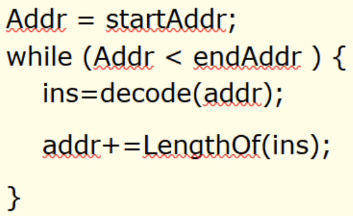
- Disassembly
- Instruction decoding/understanding
- Insertion of new code

Disassembly

- Required first step for any binary analysis or instrumentation
- 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
804964d:  89 e5        mov   %esp,%ebp
804964f:  53           push  %ebx
8049650:  83 ec 04     sub   $0x4,%esp
8049653:  eb 04        jmp   0x8049658
8049655:  e6 02 04  <junk>
8049658:  be 05000000  mov   $0x5,%esi
```

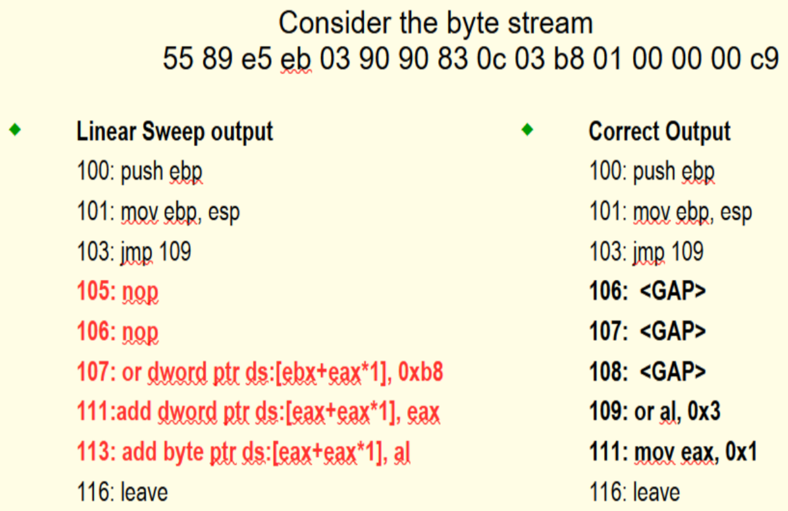
```
804964c:  55           push  %ebp
804964d:  89 e5        mov   %esp,%ebp
804964f:  53           push  %ebx
8049650:  83 ec 04     sub   $0x4,%esp
8049653:  eb 04        jmp   0x8049658
8049655:  e6 02        out 0x2, al
8049657:  04 be        add al, 0xbe
8049659:  05 00000012  add eax, 0x12000000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

- Incorrectly disassembles junk (or padding) bytes
- Confusion typically cascades past the padding, causing subsequent instructions to be missed or misinterpreted.
- Strengths: Simple, does not miss code (especially for well-behaved compilers)

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

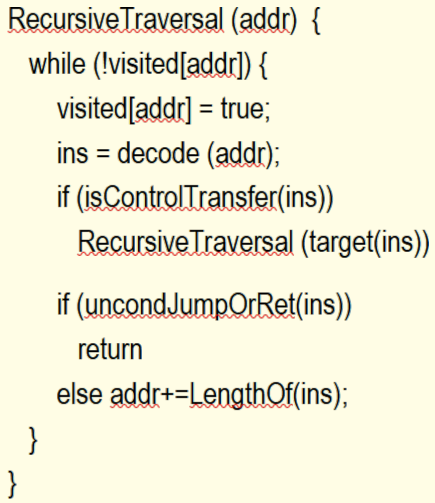
- **Linear Sweep output**

```
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



```
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);
  }
}
```

- Approach: Takes into account the control flow behavior of the program
- Strength: Not confused by embedded data
- Weakness: For indirect control transfers, the target cannot be determined statically.
  - Code reached *only* indirectly will remain unrecognized

Obfuscation against Static Disassembly

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

Code Transformation Challenges

- Code transformations change its size
  - Consider, for example, the addition of CFI or SFI checks
- This means code locations are changed
  - Control-flow targets will all be wrong, and need to be “fixed up”
- As usual, direct transfers are easier to handle
  - Their locations can be determined at transformation time
    - It is nontrivial effort in a binary instrumentation tool to fix them up, but doable

Code Transformation: Main Challenge

- Key Problem: Indirect control transfers
  - Code pointers look like (integer) data values!
    - For instance, "fptr = &f" will look like "mov eax, 0x080010b8"
- Finding the new location corresponding to 0x080010b8 isn't hard
  - No different from the handling of direct control transfer targets
- But what do we with the constant itself?
  - If 0x080010b8 is a reference to code address, then it should be replaced with the new location of the code residing at this address
  - If it is data, it should be left alone
  - How do we decide?
    - If we make a mistake, the program won't work correctly!

Code Transformation: Main Challenge

- And what if it is both?
  - Used as code address in some contexts, data in other contexts
    - static data embedded within a function may be accessed using an offset from the location of the function
    - self-examining code

Dynamic Binary Translation

Just-in-time Disassembly

- Key question: If disassembly is hard because we don't know what is code, why not wait until runtime?
- Key point: Code knows itself!
  - Valid code will only jump to valid locations
- So, delay disassembly of a code snippets until program jumps to them!
  - Code is transformed one basic block at a time

    Note: It is trivial to reliably disassemble a single basic block, which is straight-line code
- Even obfuscated code can be handled
  - No way to "hide" code: it will be found before execution!

Just-in-time Pointer Fixups

- Just as we rely on code to reveal itself, can we wait for code pointers to reveal themselves??
- Yes! If a register is used as a jump target, then its content must be a code pointer
  - Fixup code pointers just before they are used
  - Called (runtime or dynamic) address translation
- This means that we don't change any of the constants in the original code or data.
  - Guaranteed not to break code.

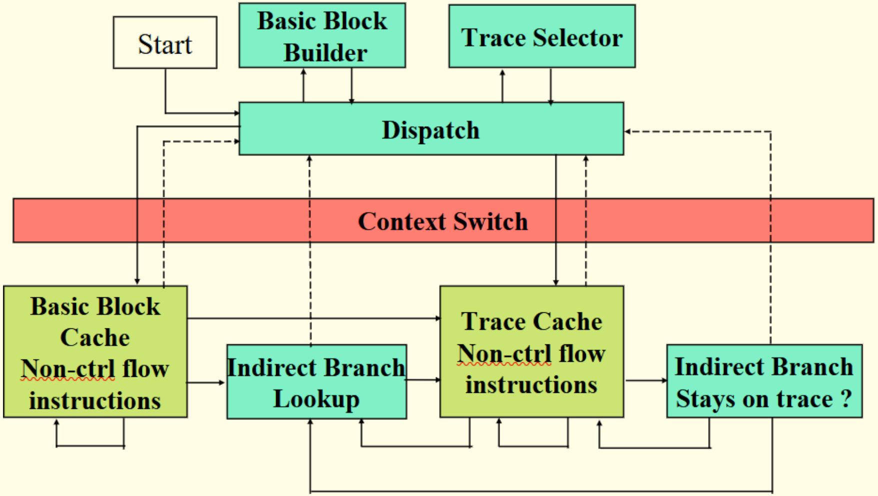
Dynamic Translation: Can it Work?

- May seem like a road to nowhere
  - Likely to be incredibly slow?
- Not necessarily:
  - Just as SFI can be competitive with hardware memory protection, DynamoRIO showed that runtime disassembly and instrumentation can be practical!

DynamoRIO

- All code is discovered at runtime, analyzed at runtime, and then rewritten ("translated") at runtime
    - Code is transformed one basic block at a time
- Only the first execution of a basic block requires analysis and rewriting.
    - Subsequent executions can use the same rewritten block.
- Control transfers occur in the last instruction of a basic block.
    - These instructions need to instrumented
        - perform address translation
        - if the target code is not already instrumented, disassemble and instrument it
- Non-control-transfer instructions can be executed natively

RIO System Infrastructure


Start
Basic Block Builder
Trace Selector
Dispatch
Context Switch
Basic Block Cache Non-ctrl flow instructions
Indirect Branch Lookup
Trace Cache Non-ctrl flow instructions
Indirect Branch Stays on trace ?

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.
  - 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

Address Translation

- 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)
  - 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)
- 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.

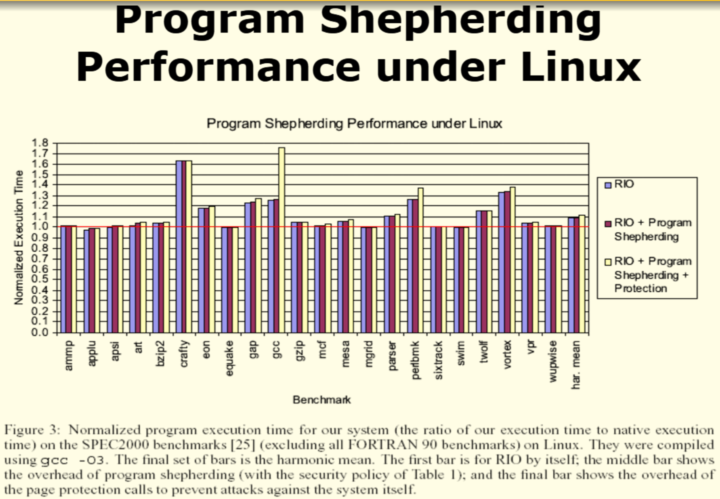
Transparency Issues

- 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: DynamoRIO's use of libraries shouldn't be visible to the application
  - Thread transparency: DynamoRIO does not create its own threads
  - Stack transparency: DynamoRIO does not touch the application stack
  - Context Translation: On exceptions, faulting address shouldn't be the translated address.
  - Address space transparency: DynamoRIO should not leak its location

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



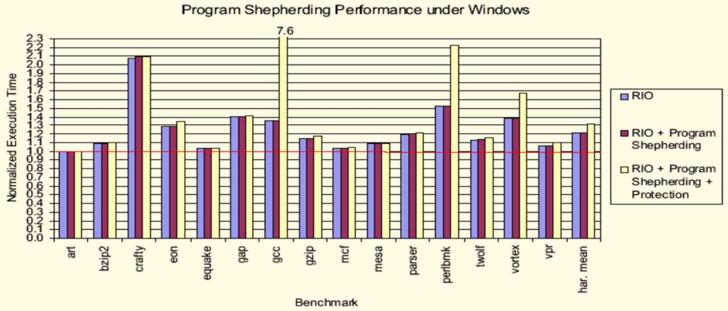
Program Shepherding Performance under Linux

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

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 `cl /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 long-running 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 (Continued)

- 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

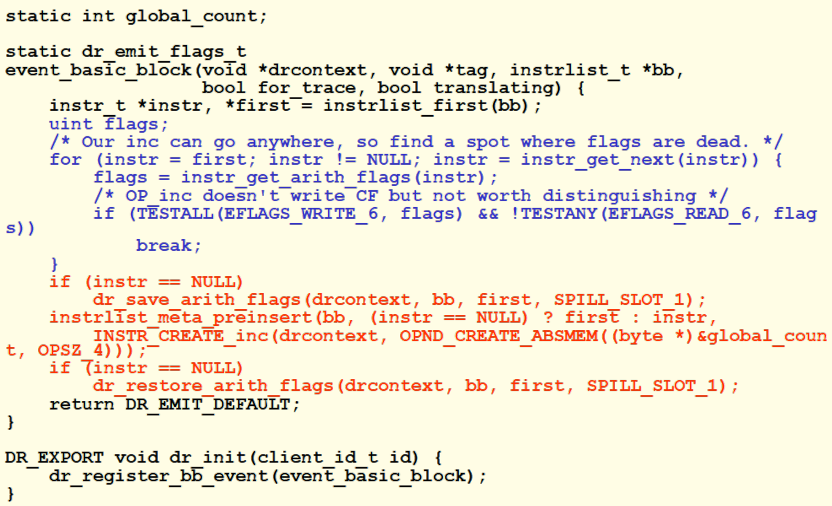
```
static int bbcount;

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, IPOINT_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, flag
s))
            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_coun
t, 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);
}
```

Static Binary Rewriting

Drawbacks of Dynamic Binary Rewriting

- High overhead
- The need for a large and complex runtime system
  - Often, leaves code cache to be writable to reduce overhead
  - Even otherwise, a large runtime footprint means a large attack surface.

Static Binary Instrumentation: Challenges

- Robust static disassembly
  - Including low-level libraries and hand-written assembly
- Static instrumentation without breaking complex code
  - Fixing up indirect control transfers
  - Fixing up direct transfers
  - Tolerating disassembly errors
- Secure instrumentation
  - Ensure instrumentation of *all* code
  - Ensure that added security checks cannot be bypassed

Static Disassembly: BinCFI approach

- Take advantage of the fact that the presence of data in code is rare
  - Use linear disassembly, followed by error detection and correction
  - Error detection is based on control flow consistency
- Tolerate disassembly errors:
  - Ensure that if data is disassembled as code, that does not cause misbehavior of instrumented code

Pointer Fixup

- Direct control transfers: Instrument assembly code
  - "Reassemblable disassembly:" Disassembled code can be reassembled into binary with full preservation of behavior
  - Use labels so that the assembler can figure out actual instruction offsets etc.
- Indirect control transfers:
  - Static analysis to discover all possible code pointers
    - Conservative approach: may include non-code pointers, but cannot leave out legitimate ones
  - Runtime address translation to translate any of these
    - Provides most transparency benefits of dynamic translation techniques

Safe and Secure Instrumentation

- Make a second copy of code and instrument it
  - It is OK if you disassemble and instrument data, as the original data is left in place
- Control-flow integrity ensures that only disassembled code is instrumented
  - If some code is somehow missed, it leads to failure rather than security violation
- CFI also protects all the added instrumentation
  - CFI disallows “jumping past instrumentation”

BinCFI Results

- Supports large and low-level COTS (“stripped”) binaries
  - glibc, Firefox, Adobe Reader, gimp, etc.
    - Over 300MB of (intel 32-bit) binaries in total.
- Eliminates 99% of control-flow targets and 93% of possible gadgets
  - Remaining gadgets provide very limited capability
- Good performance while providing full transparency
  - About 10% overhead on CPU-intensive C-benchmarks, somewhat higher for C++ programs

Static Instrumentation: Further Performance Improvements...

- Most of BinCFI's overhead comes from runtime code pointer translation
- *Question:* Can we avoid this runtime translation?
    - Requires code pointers to be translated at instrumentation time

Static Instrumentation: Further Performance Improvements...

- Most of BinCFI's overhead comes from runtime code pointer translation
- *Question:* Can we avoid this runtime translation?
  - Requires code pointers to be translated at instrumentation time
- *Yes:* For 64-bit position-independent binaries
  - Almost all code on modern Linux distributions falls in this category
  - Pointers are all explicitly identified in these binaries
    - but there is no information on which of these point to code

Static Instrumentation: Further Performance Improvements...

- Most of BinCFI's overhead comes from runtime code pointer translation
- *Question:* Can we avoid this runtime translation?
  - Requires code pointers to be translated at instrumentation time
- *Yes:* For 64-bit position-independent binaries
  - Almost all code on modern Linux distributions falls in this category
  - Pointers are all explicitly identified in these binaries
    - but there is no information on which of these point to code
- Approach: Develop static analysis to distinguish code and data pointers
  - Relies on detailed instruction semantics

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