Motivation

Why Binary Instrumentation (rewriting)?
- Forms the basis of many software security defenses.
- Binary code broadly available

*Dynamic binary instrumentation (DBI) is the de facto choice.* (easy-to-use API, can handle large binaries, non-bypassable)

Issues with DBI: **performance**
- Start up overhead
- System call
- State maintenance

static binary rewriting can do much better in these situations!
Talk Outline

Background
System Overview and Highlights
Instrumentation Applications
Evaluation
Conclusion
Static Binary Disassembly (Background)

Binary disassembly is a critical issue for SBI. Previous work has shown various solutions:

- **ILR [S&P ‘12]:** IDA-pro + objdump
- **Reins [ACSAC ‘12]:** IDApython
- **binCFI [USENIX ‘13]:** objdump+static analysis
- **CCFIR [S&P ‘13]:** IDA-pro + relocation
Issues in Current SBI System

Cannot guarantee all code transformation, ie. dynamically loaded libraries cannot be found at static rewriting time.

- dlopen? configuration/plugin
- environment variable? LD_LIBRARY_PATH=...

Fact of dynamic libraries: wireshark (45/144), gedit(17/74), acroread9 (21/82), gimp-2.6 (151/206)

No general purpose SBI system

- Control Flow Integrity (CFI)
- Software based Fault Isolation (SFI)
System Overview and Highlights

PSI is a general purpose binary rewriting platform that has the advantages:

- **All program transformation**
  - transform exes, libs and loader
- **Secure instrumentation**
  - instrumentation code is non-bypassable
- **Versatile, easy-to-use API**
  - low level API + high level API
- **Good performance**
  - 7x ~ 13x lower overhead on many real world programs
All Program Transformation

**PSI supports offline rewriting and on-the-fly rewriting**

- Offline rewriting
  - psi_rewriter -t instrument.src /bin/ls -o ./ls_trans
Offline Rewriting
All Program Transformation

PSI supports offline rewriting and on-the-fly rewriting

- Offline rewriting
  - psi_rewriter -t instrument.src /bin/ls -o ./ls_trans

Offline rewriting can transform known dependencies, but some dependencies might not be known until runtime.
All Program Transformation

**PSI supports offline rewriting and on-the-fly rewriting**

- **Offline rewriting**
  - `psi_rewriter -t instrument.src /bin/ls -o ./ls_trans`

  Offline rewriting can transform known dependencies, but some dependencies might not be known until runtime.

- **On-the-fly rewriting**
  - `psi_loader -t instrument.src -- /bin/ls`  
    *(invokes psi_rewriter for each library)*

  On-the-fly rewriting transforms the binary and all dependent libraries at runtime
On-the-fly Binary Rewriting

Implementation mechanism:

- step 1: send msg to daemon
- step 2: if lib is transformed goto 5
- step 3: perform rewriting
- step 4: ack msg to psi loader
- step 5: perform loading
Secure Instrumentation

Why? Security Instrumentation become useless if can be bypassed

How? All indirect transfer => Table Lookup

/* original code */

jump *%ecx

/* translated code */

mov %eax, %gs:0x40
mov %ecx, %eax
jump table_lookup
Secure Instrumentation

- All indirect branch targets are checked by table lookup
- Control flows application code cannot interfere instrumentation
Secure Instrumentation

Subverted indirect branch to inline instrumentation code is defeated

table lookup fail
Secure Instrumentation

Subverted indirect branch to instrumentation client is defeated

empty table in client...
Instrumentation Applications

- Basic block counting (in-line)
- Shadow stack (in-line + out-of-line)
- System call policy enforcement
- Library load policy enforcement
Basic Block Counting

/******* Binary Rewriting code *******/

foreach bb in getBBs() {
    found = false
    foreach insn in bb {
        if isTest(insn) or isCmp(insn) {
            found = true
            ins_snippet(insn, BEFORE, opt)
            break
        }
    }
    if !found
        ins_snippet(bb, BEGIN, unopt)
}
Basic Block Counting

/***** Added Instrumentation (Inline) ****/

unopt = "mov %eax, TS_0;
lahf;
incl TS_1;
sahf;
mov TS_0, %eax"

opt = "incl TS_1"
Shadow Stack

A classic security defense against buffer overflow and return-oriented programming (ROP)

Main idea:

- maintain a 2nd stack containing return addresses
- All calls push return address both stacks
- All returns check the consistency of two stacks
Shadow Stack

/* Instrumentation (Inline )**/

/* shadow stack pointer is
 * stored in TS_2 */
chk_init_shadowstk = "
    cmp $0x0, TS_2;
    jnz L001;
    call $alloc_stack;
L001: ";

push_shadowstk = "
    mov %eax, TS_0;
    mov %ebx, TS_1;
    subl $4, TS_2;
    mov TS_2, %eax;
    mov (%esp), %ebx;
    mov %ebx, (%eax);
    mov TS_0, %eax;
    mov TS_1, %ebx;"

/***** Out-of-line Instrumentation code *******

void check_return(Context *ctxt) {
    uint shadow_sp = ctxt->TS[2];
    uint ret = getmem(ctxt->ESP);
    while (!empty(shadow_sp)){
        if (pop(shadow_sp) == ret) {
            ctxt->TS[2] = shadow_sp;
            return;
        }
    }
    abort();
}

/******* Binary Rewriting code **************/

foreach insn in getInsns()
    if isCall(insn) {
        ins_snippet(insn, BEFORE, chk_init_shadowstk)
        ins_snippet(insn, BEFORE, push_shadowstk)
    }
    else if isRet(insn)
        ins_call(insn, AFTER_CALL, check_return)
Shadow Stack

/**In-line Instrumentation**/

/* shadow stack pointer is * stored in TS_2 */
chk_init_shadowstk = "
    cmp $0x0, TS_2;
    jnz L001;
    call $alloc_stack;
L001: ";

push_shadowstk = "
    mov %eax, TS_0;
    mov %ebx, TS_1;
    subl $4, TS_2;
    mov TS_2, %eax;
    mov (%esp), %ebx;
    mov %ebx, (%eax);
    mov TS_0, %eax;
    mov TS_1, %ebx;"

/***** Out-of-line Instrumentation code *******/

void check_return(Context *ctxt) {
    uint shadow_sp = ctxt->TS[2];
    uint ret = getmem(ctxt->ESP);
    while (!empty(shadow_sp)){
        if (pop(shadow_sp) == ret) {
            ctxt->TS[2] = shadow_sp;
            return;
        }
    }
    abort();
}

/******* Binary Rewriting code *************/

foreach insn in getInsns() {
    if isCall(insn) {
        ins_snippet(insn, BEFORE, chk_init_shadowstk)
        ins_snippet(insn, BEFORE, push_shadowstk)
    } else if isRet(insn)
        ins_call(insn, AFTER_CALL, check_return)
Shadow Stack

**In-line Instrumentation**

/* shadow stack pointer is stored in TS_2 */
chk_init_shadowstk = "
  cmp $0x0, TS_2;
  jnz L001;
  call $alloc_stack;
L001: ";

push_shadowstk = "
  mov %eax, TS_0;
  mov %ebx, TS_1;
  subl $4, TS_2;
  mov TS_2, %eax;
  mov (%esp), %ebx;
  mov %ebx, (%eax);
  mov TS_0, %eax;
  mov TS_1, %ebx;
"

**** Out-of-line Instrumentation code *******/

void check_return(Context *ctxt) {
  uint shadow_sp = ctxt->TS[2];
  uint ret = getmem(ctxt->ESP);
  while (!empty(shadow_sp)){
    if (pop(shadow_sp) == ret) {
      ctxt->TS[2] = shadow_sp;
      return;
    }
  }
  abort();
}

******* Binary Rewriting code *************/

foreach insn in getInsns()
  if isCall(insn) {
    ins_snippet(insn, BEFORE, chk_init_shadowstk)
    ins_snippet(insn, BEFORE, push_shadowstk)
  }
  else if isRet(insn)
    ins_call(insn, AFTER_CALL, check_return)
/* Shadow Stack */

/**In-line Instrumentation**/  
/* shadow stack pointer is  
 * stored in TS_2 */
chk_init_shadowstk = "  
      cmp $0x0, TS_2;  
      jnz L001;  
      call $alloc_stack;  
L001: ";  

push_shadowstk = "  
      mov %eax, TS_0;  
      mov %ebx, TS_1;  
      subl $4, TS_2;  
      mov TS_2, %eax;  
      mov (%esp), %ebx;  
      mov TS_0, %eax;  
      mov TS_1, %ebx;"

/***/

/***** Out-of-line Instrumentation code ******/

void check_return(Context *ctxt) {
    uint shadow_sp = ctxt->TS[2];
    uint ret = getmem(ctxt->ESP);
    while (!empty(shadow_sp)){
        if (pop(shadow_sp) == ret) {
            ctxt->TS[2] = shadow_sp;
            return;
        }
    }
    abort();
}

/******** Binary Rewriting code **************/

foreach insn in getInsns()
    if isCall(insn) {
      ins_snippet(insn, BEFORE, chk_init_shadowstk)
      ins_snippet(insn, BEFORE, push_shadowstk)
    }  
    else if isRet(insn)
      ins_call(insn, AFTER_CALL, check_return)
Evaluation

Basic block counting in SPEC2006
Shadow stack
Micro benchmark
Real world applications
Basic Block Counting for SPEC2006
Shadow Stack

The graph compares the performance overhead of Shadow Stack (PSI) and ROPdefender (Pin) across various benchmarks. Each bar represents the percentage overhead for a specific benchmark, with PSI shown in red and Pin in blue. The x-axis lists the benchmarks, and the y-axis shows the percentage overhead. The benchmarks include 400.perlbench, 401.bzip2, 403.gcc, 429.mcf, 445.gobmk, 456.hmmer, 462.libquantum, 464.h264ref, 473.astar, 473.libquantum, 473.mlfr, 473.milc, 473.namd, 473.sphinx3, and others.
Micro Benchmark: Imbench
## Real World Application Evaluation

<table>
<thead>
<tr>
<th>Program</th>
<th>PSI</th>
<th>DynamoRIO</th>
<th>Pin</th>
<th>Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>coreutils</td>
<td>97%</td>
<td>1922%</td>
<td>3509%</td>
<td>coreutils testsuite</td>
</tr>
<tr>
<td>gcc</td>
<td>63%</td>
<td>1376%</td>
<td>10250%</td>
<td>openssh compilation</td>
</tr>
<tr>
<td>apt-get update</td>
<td>2%</td>
<td>326%</td>
<td>411%</td>
<td>run cmd 5 times and get median.</td>
</tr>
<tr>
<td>latex</td>
<td>51%</td>
<td>185%</td>
<td>1806%</td>
<td>compile 17KB tex file into dvi</td>
</tr>
<tr>
<td>python</td>
<td>33%</td>
<td>85%</td>
<td>96%</td>
<td>pystone 1.1 bench</td>
</tr>
<tr>
<td>......</td>
<td>.....</td>
<td>.....</td>
<td>.....</td>
<td>.....</td>
</tr>
<tr>
<td>average</td>
<td>53%</td>
<td>887%</td>
<td>3412%</td>
<td></td>
</tr>
</tbody>
</table>
Related Work

- **Static Binary Rewriting**
  - Most previous works require compiler support, symbol, debugging, or relocation information: Native Client [S&P 2009], SFI [Security 2006], G-Free [ACSAC 2010], CFI [TISSEC 2009], XFI [OSDI 2006], CCFIR [S&P 2013]...
  - Systems targeting COTS binaries
    - SecondWrite [EuroSys 2013] emphasizes binary analysis; no non-bypassable instrumentation support
    - Reins [ACSAC 2012]: Windows executables, not library support

- **Dynamic Binary Instrumentation**
  - Frameworks:
    - Pin [PLDI 2005], DynamoRIO [PhD Thesis 2004]
  - Security Instrumentation:
    - Program Sheperding [Security 2002], libdetox [VEE 2011]
PSI is a general purpose binary rewriting platform that has the advantages:

- **All program transformation**
  - transform exes, libs and loader
- **Secure instrumentation**
  - instrumentation code is non-bypassable
- **Versatile, easy-to-use API**
  - low level API + high level API
- **Good performance**
  - 7x ~ 13x lower overhead on many real world programs

Questions?