The Effect of Instruction Padding on SFI Overhead

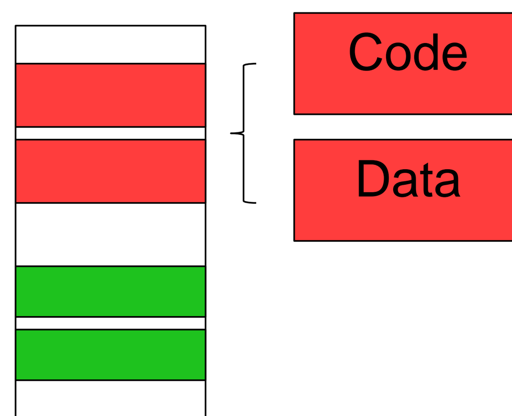
Navid Emamdoost, Stephen McCamant
Department of Computer Science and Engineering, University of Minnesota, Twin Cities

1. Motivation

- Improving current isolation mechanism performance
  - Guaranteeing same level of security
- Native Client
  - Software-based Fault Isolation implementation for CISC architectures
  - Instruction padding to enforce security policies
  - Padding imposes runtime overhead
  - We changed the padding scheme
  - Updated validator accordingly

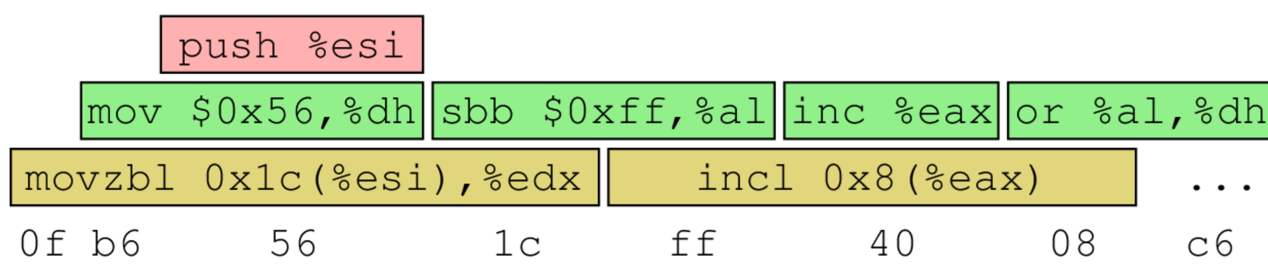
2. Software-based Fault Isolation

- Applications may incorporate independently developed modules
  - Operating System: Add new file system
  - Database Management System: User-defined data type
- Problem with extensions
  - Security
  - Reliability
- Solution: Isolation
- Load untrusted extension into its own fault domain
  - Code Segment
  - Data Segment
- Security Policy:
  - No code is executed outside of fault domain
  - No data is changed outside of fault domain



3. Google Native Client (NaCl)

- An SFI implementation for CISC and RISC architectures
- Allows execution of untrusted C/C++ code in Chrome browser
- Gives performance of native code to browser plugins
- On CISC architectures, it incorporates instruction padding to enforce an address layout invariant and restrict control flow
- Problem with variable size instructions in CISC architectures:



4. Instruction Padding to Enforce Security Policies

- Unsafe instructions
  - `jmp *%ecx`
  - `mov $0x1b80,(%ecx)`
- Padding Scheme
  - Divide memory into 32-byte *Bundles* (red boxes in the following listing)
  - Target of jumps placed at the beginning of bundles (type 1)
  - Call instructions placed at the end of bundles (type 2)
  - No instruction is allowed to cross bundle boundary (type 3)

```
1060440:    83 c8 01                 or $0x1,%eax
                                     ...
106045c:    85 d2                    test    %edx,%edx
106045e:    66 90                    xchg    %ax,%ax
1060460:    0f 88 9a 01 00 00        js 1060600
1060466:    8b 01                    mov (%ecx),%eax
                                     ...
1060477:    83 fa ff                 cmp $0xffffffff,%edx
106047a:    8d b6 00 00 00 00        lea 0x0(%esi),%esi
1060480:    c7 04 38 84 06 03 11 movl    $0x11030684,(%eax,%edi,1)
1060487:    8d 47 fc                 lea -0x4(%edi),%eax
                                     ...
10ee700:    55                       push    %ebp
                                     ...
10ee70d:    8d b4 26 00 00 00 00 lea 0x0(%esi,%eiz,1),%esi
10ee714:    8d bc 27 00 00 00 00 lea 0x0(%edi,%eiz,1),%edi
10ee71b:    e8 a0 72 f5 ff           call    10459c0
10ee720:    89 ec                    mov %ebp,%esp
                                     ...
10ee724:    83 e1 e0                 and $0xffffffe0,%ecx
10ee727:    ff e1                    jmp *%ecx
                                     ...
```

- Padding side effects
  - Wasting CPU cycles by executing NOP
  - Reducing code density and, as a result, increasing instruction cache misses
- Type 3 padding is conservative
  - The intention is to eliminate invalid or unwanted instructions
  - Idea: we can allow a cross-bundle instruction if we make sure no unsafe instruction is interpretable from the crossing point
  - Challenge: decide which NOP can be removed while ensuring the security policies are still enforced

5. Cross Bundle Instruction NaCl

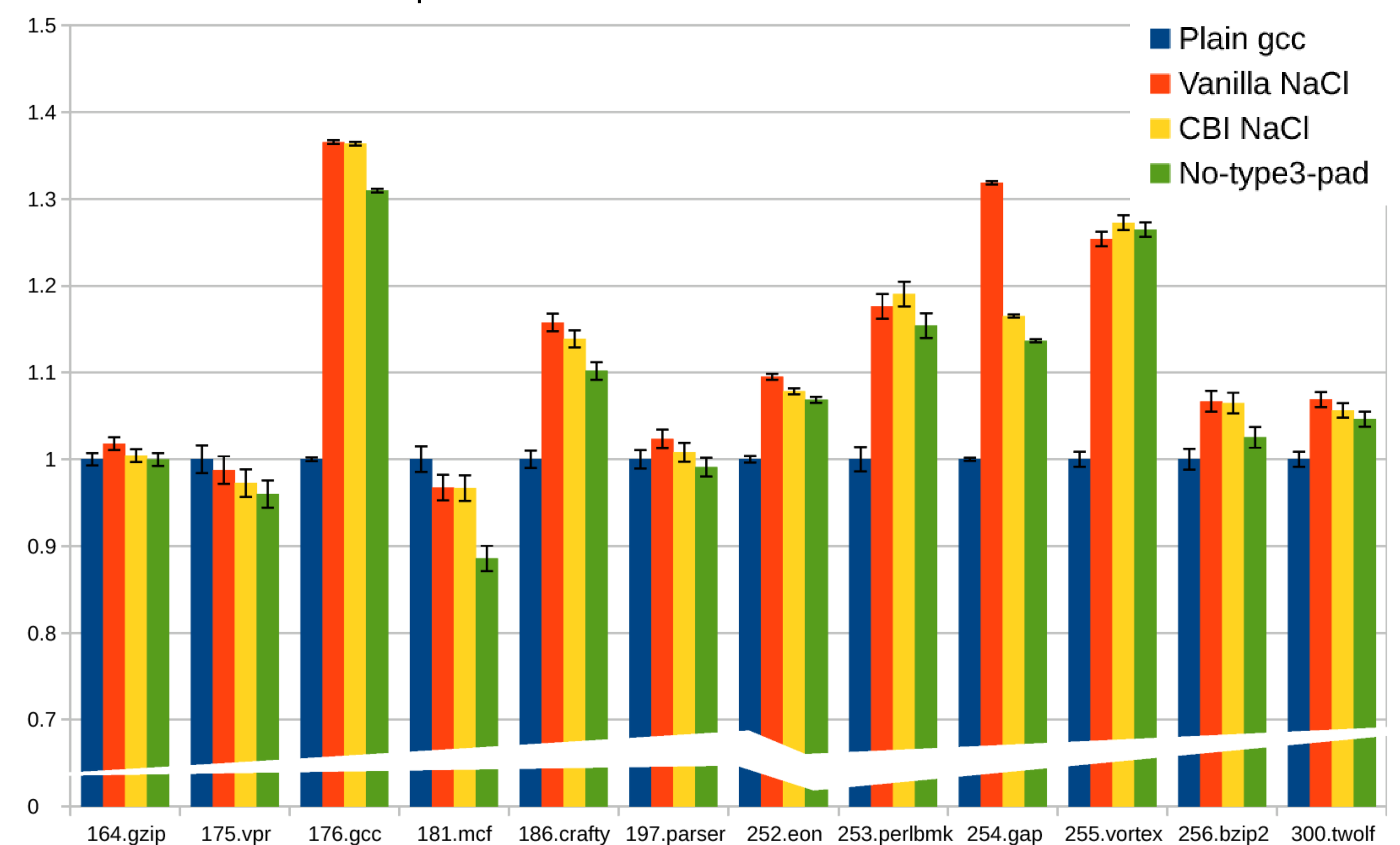
- Change padding scheme
  - Allow instructions to cross the bundle boundaries as long as no unsafe instruction stream encountered
  - Algorithm:
    - For each padding do the following:
      1. Set padding size to zero
      2. Assemble the binary
      3. If the validator fails on the binary, increase padding size by one
      4. Do the steps (2) and (3) until either validator succeeds or padding size reaches the original size
- Update the validator accordingly
  - We are allowing cross-bundle instructions in the binary
  - Validator must check no unsafe instructions are reachable
  - Multipass Validator: Start validation process from every cross point
    - Every bundle start
  - This way we can make sure every reachable address represents a valid instruction
  - We proved multipass validator correctness in Coq
    - Based on the *RockSalt* paper [G. Morrisett et al, PLDI 2012]

6. Evaluation

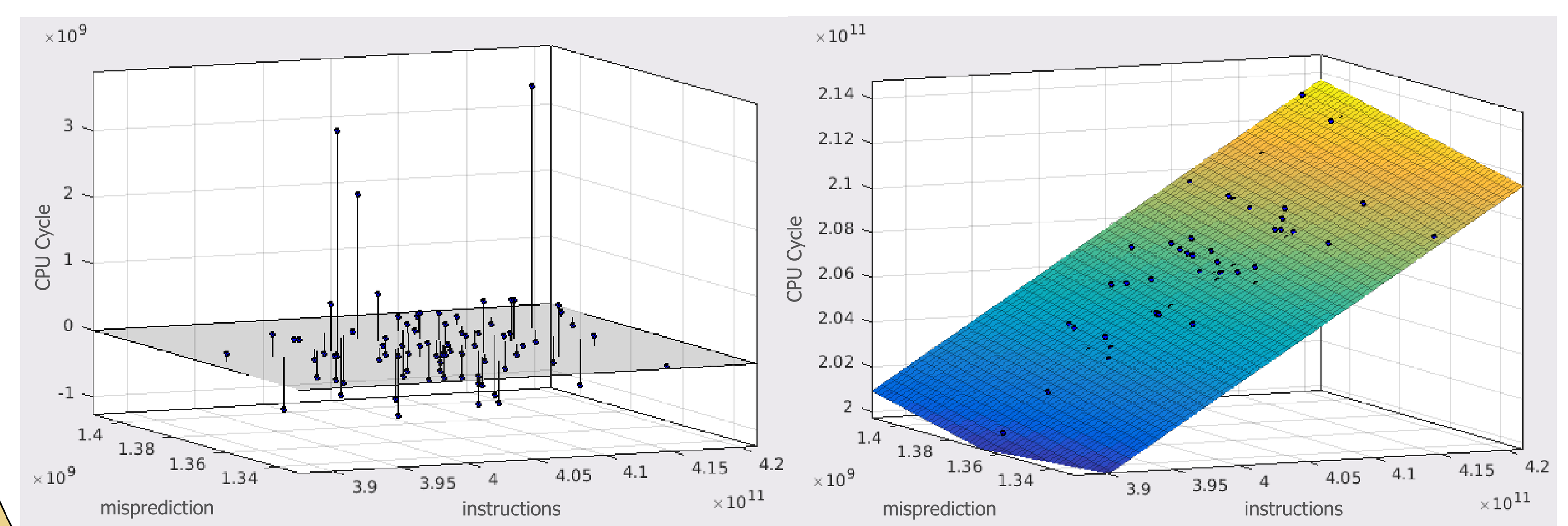
- We implemented our changes into GNU Assembler and NaCl validator
- Used SPECint CPU2000 as benchmark
- Number of instructions executed

Benchmark	Vanilla NaCl	CBI NaCl	decrease %
gzip	906e+9	896e+9	1.1%
vpr	200,395e+6	200,373e+6	<0.1%
gcc	175e+9	173e+9	1.4%
mcf	55e+9	54e+9	2.2%
crafty	226e+9	223e+9	1.2%
parser	342e+9	338e+9	1.1%
eon	215,899,844e+3	215,899,808e+3	<0.1%
perlbmk	400e+9	396e+9	0.9%
gap	433e+9	428e+9	1.1%
vortex	380e+9	372e+9	1.9%
bzip2	552e+9	533e+9	3.3%
twolf	351e+9	346e+9	1.3%

- Normalized runtime comparison



- Investigation on perlbmk anomaly
  - Generate multiple samples with randomized layout
  - Monitor the execution under OProfile
  - Correlation between indirect branch misprediction and CPU Cycle



7. Conclusion and Future Work

- We proposed more permissive padding policy
- Proved it is as secure as vanilla NaCl
- This optimization leads to decrease in the number of instructions executed and modest saving of averaging 1.5% in execution time
- Future work
  - Extending the CBI idea to x86_64 architecture
  - Replacing the greedy pad removal with a dynamic programming one