Strato: A Retargetable Framework for Low-Level Inlined Reference Monitors

Bin Zeng, Gang Tan, Úlfar Erlingsson
Lehigh University Google Inc.

USENIX Security 2013
@Washington DC, USA

Attacks

- How attacks happen
  - Arrive as user input through a communication channel
  - Trigger pre-existing bugs
  - Take over program executions
- Attack vector
  - Mobile code, untrusted extensions
  - Memory corruption attacks [StackSmash]
  - Return Oriented Programming [ROP]

Existing Countermeasures

- Data Execution Protection [DEP]
- Address Space Layout Randomization [PaX]
- Program Shepherding [Shepherding]
- Inlined Reference Monitors [IRM]
  - Control Flow Integrity [CFI, XFI, HyperSafe]
  - Software-based Fault Isolation [Pittsfield, Native Client]

Inlined Reference Monitors (IRMs)

- IRM: embed security checks in programs

- Well-established against various attacks
  - E.g., buffer overflows, Return-Oriented Programming attacks

Inlined Reference Monitors (IRMs)

- CFI (Control-Flow Integrity): checks control flow
- SFI (Software-based Fault Isolation) also checks memory reads and writes

- Example: Google's Native Client
  - **Verifiable** machine code plugins for browsers

However, Most IRM Implementations are Low-Level

- Binary rewriting, assembly instrumentation,…
- Implementations
  - Tightly coupled with architectures
  - Hard to reuse
- For example, Native Client (NaCl) has multiple implementations
  - x86-32; x86-64

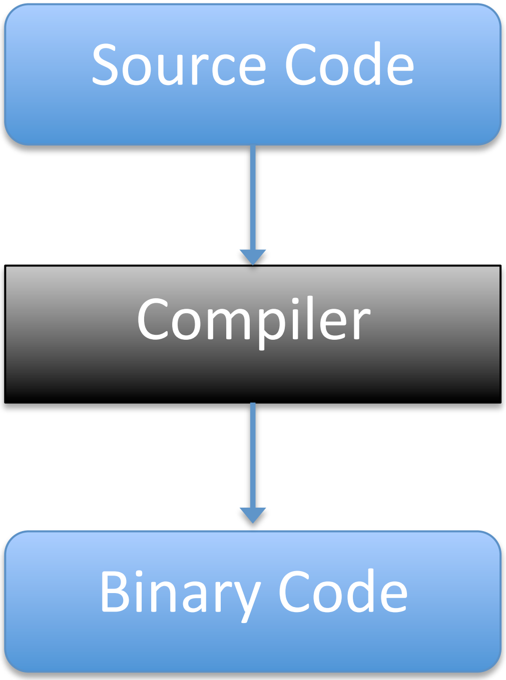
Our General Idea

- Perform IRM rewriting at an Intermediate-Representation (IR) level
  - Use an IR that is largely architecture-independent (in particular, LLVM IR)
- Benefits
  - Reuse transformations among architectures
  - IR is amenable to optimizations
- Retain verifiability of low-level code

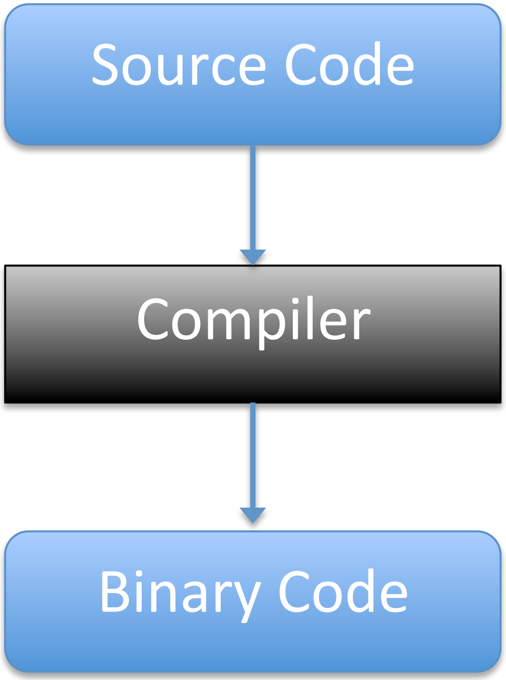
Challenges of IR-level Rewriting

- Compiler transformations after the IR can invalidate security assumptions
- Have to trust the compiler back-end from IR to low-level code
  - TCB Bloat

Are Compilers Trustworthy?



Compilers are Buggy



Compilers are Buggy

- Compilers have a huge code base
  - GCC 4.8 has more than 7.3 million lines of code
- Csmith found 300+ unknown bugs [PLDI '11]
- LLVM has a steady bug rate

Buggy Compiler Optimizations

Any sufficiently optimizing compiler is indistinguishable from magic.

-- Paraphrasing Arthur C. Clarke

Compiler Optimizations

- Compiler optimizations invalidate security assumptions
- They only care about functional semantics
- Security properties are often non-functional

Research Question

- How to do IRM rewriting at the IR level, and preserve low-level security?

- Our paper's contribution:
  - Strato: a IRM-implementation framework that performs IR-level rewriting and preserves low-level security

Key Challenge

- Challenge: after checks are inserted at the IR level, backend transformations may invalidate security – if all data memory is untrusted

Before register allocation

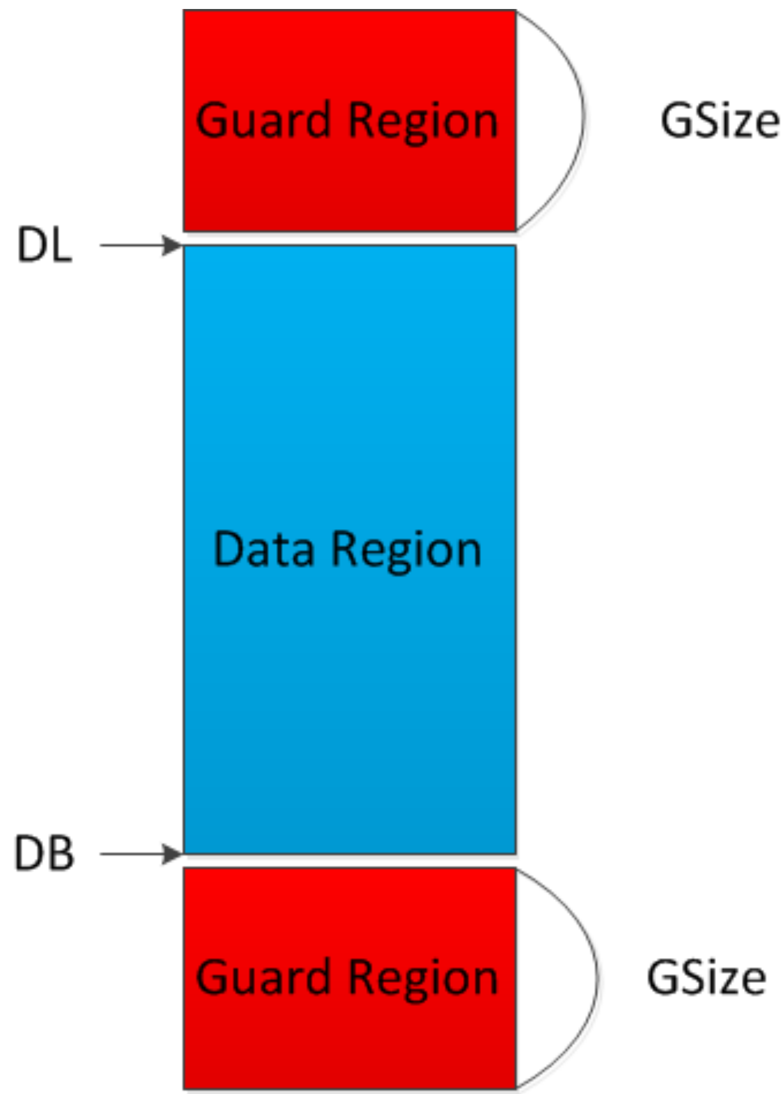
```
ptr.safe = check(ptr)
tmp = load *ptr.safe
store v, *ptr.safe
```

After register allocation

```
ptr.safe = check(ptr)
tmp = load *ptr.safe
store ptr.safe, *stack_loc
ptr.safe2 = load *stack_loc
store v, *ptr.safe2
```

Register spilling

Attack Model



Guard region is mprotected

Anything from memory is untrusted

Our Idea for Addressing the Problem

- Insert more-than-enough checks at the IR level
- Attach constraints to checks to encode conditions that might be invalidated by the compiler
- After compiler transformations, perform constraint checking at the low level
  - Remove checks iff constraints are still valid
  - If a compiler transformation invalidates a constraint, then the check is left intact for security

Let's go through an example next

Uninstrumented IR Code

```
entry:
        tmp = 0
        if(v > 47) goto then
else:
        tmp = load *ptr
        goto end
then:
        store v, *ptr
end:
        ret tmp
```

Instrumented and Optimized IR

```
entry:
    ptr.safe = check(ptr) // check1
    tmp = 0
    if(v > 47) goto then
else:
    ptr.safe1 = check(ptr.safe) // check2
    # noSpill(ptr.safe, check1, check2)
    tmp = load *ptr.safe1
    goto end
then:
    ptr.safe2 = call check(ptr.safe) // check3
    # noSpill(ptr.safe, check1, check3)
    store v, *ptr.safe2
end:
    ret tmp
```

Security checks

Constraints

Original code

Redundant Check Elimination

After Constraint Checking

```
entry:
    ptr.safe = check(ptr) // check1
    tmp = 0
    if(v > 47) goto then
else:
    ~~ptr.safe1 = check(ptr.safe) // check2~~
    # noSpill(ptr.safe, check1, check2)
    tmp = load *ptr.safe
    goto end
then:
    ptr.safe2 = call check(ptr.safe) // check3
    # noSpill(ptr.safe, check1, check3)
    store v, *ptr.safe2
end:
    ret tmp
```

Assume ptr.safe not spilled between check1 and check2, but spilled between check2 and check3

Another Example: Uninstrumented IR Code

```
x = gep p, 0, 0
tmp1 = load *x
y = gep p, 0, 1
tmp2 = load *y
sum = add tmp1, tmp2
ret sum
```

Instrumented and Optimized IR

```
p.safe = check(p) // check1
x = gep p.safe, 0, 0
x.safe = check(x) // check2
# noSpill(p.safe, check1, check2)
# sizeof(struct s)*0 + sizeof(long)*0 < GZSize
tmp1 = load *x.safe
y = gep p.safe, 0, 1
y.safe = check(y) // check3
# noSpill(p.safe, check1, check3)
# sizeof(struct s)*0 + sizeof(long)*1 < GZSize
tmp2 = load *y.safe
sum = add tmp1, tmp2
ret sum
```

Sequential Memory Access Optimization

After Constraint Checking

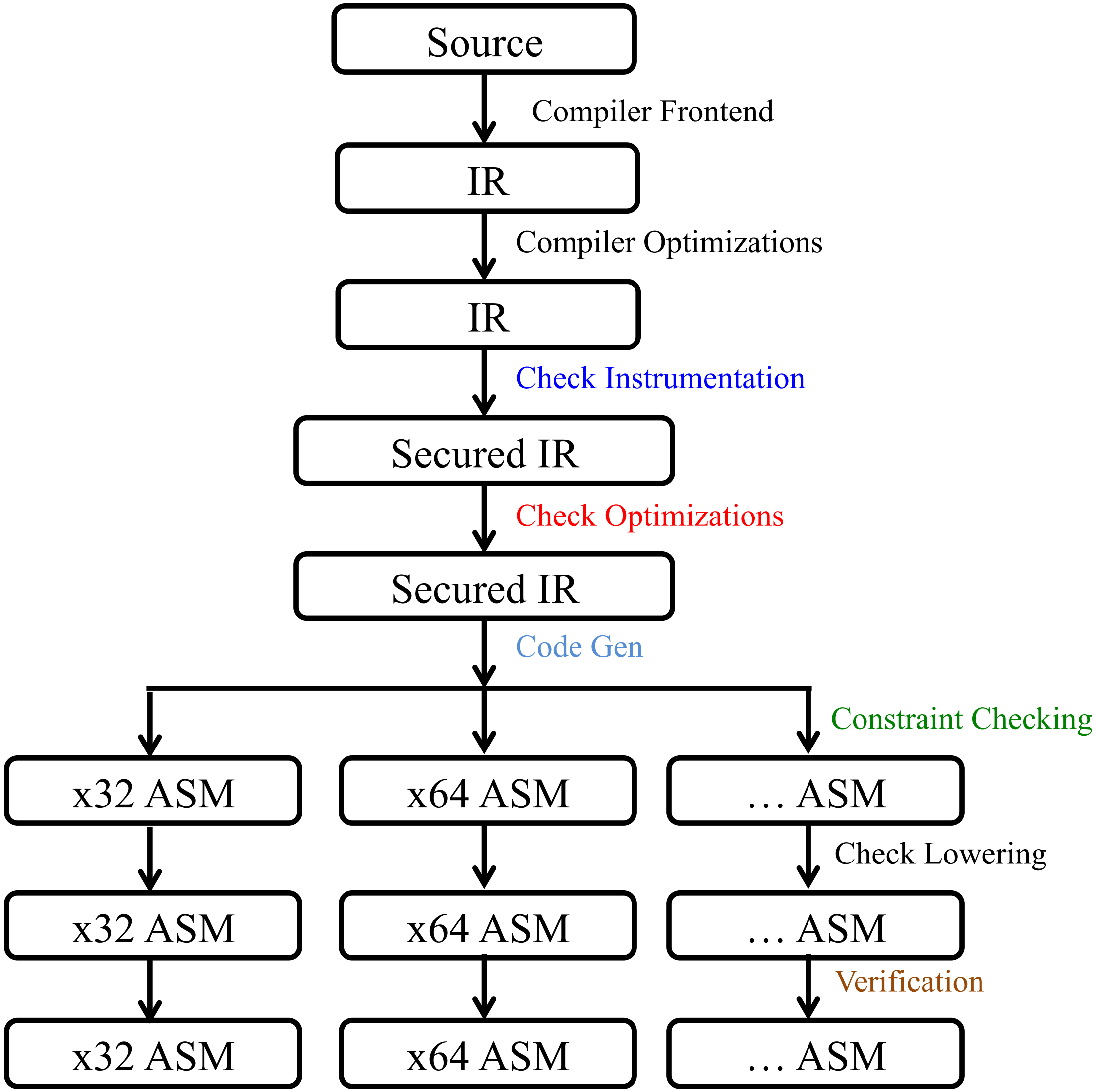
```
p.safe = check(p) // check1
x = gep p.safe, 0, 0
~~x.safe = check(x) // check2~~
# noSpill(p.safe, check1, check2)
# sizeof(struct s)*0 + sizeof(long)*0 < GSize
tmp1 = load *x.safe
y = gep p.safe, 0, 1
~~y.safe = check(y) // check3~~
# noSpill(p.safe, check1, check3)
# sizeof(struct s)*0 + sizeof(long)*1 < GSize
tmp2 = load *y.safe
sum = add tmp1, tmp2
ret sum
```

Assume (1) ptr.safe not spilled between check1 and check2, or check1 and check3 (2) offsets less than guard-zone size

Strato: Retargetable IRMs

- Instrumentation at intermediate representation level, i.e. LLVM IR
  - IR-level checks
- Optimizations of security checks and attach constraints
- Constraint-checking before lowering
  - If a constraint holds, remove the check
  - Otherwise, lower the IR-level check to machine code
- Verification at the low level
  - Remove everything else outside the TCB (including constraint checking)

The Architecture of Strato



Benefits

- Retargetable
  - Easy to port to other architectures
- Enable optimizations
  - Structured information at the IR level
  - Static Single Assignment form
- Code reuse
  - Instrumentation and optimizations can be shared among various architectures

The Implementation of Strato

- Two policies: CFI & SFI
- Instrumentation
  - Function passes into the end LLVM pipeline
- Optimizations
  - Redundant Check Elimination
  - Sequential Memory Access Optimization
  - Loop-based Check Optimization
  - Optimizations attach constraints
- Constraint checking
- Range analysis (interval analysis) based verifier

Verification

- Based on CCS paper [CCS' 11]
- After all the optimizations, constraint checking, a verifier verifies the final result in assembly code
- Removes everything before out of TCB
- Based on range analysis
- Found a few bugs in our implementation

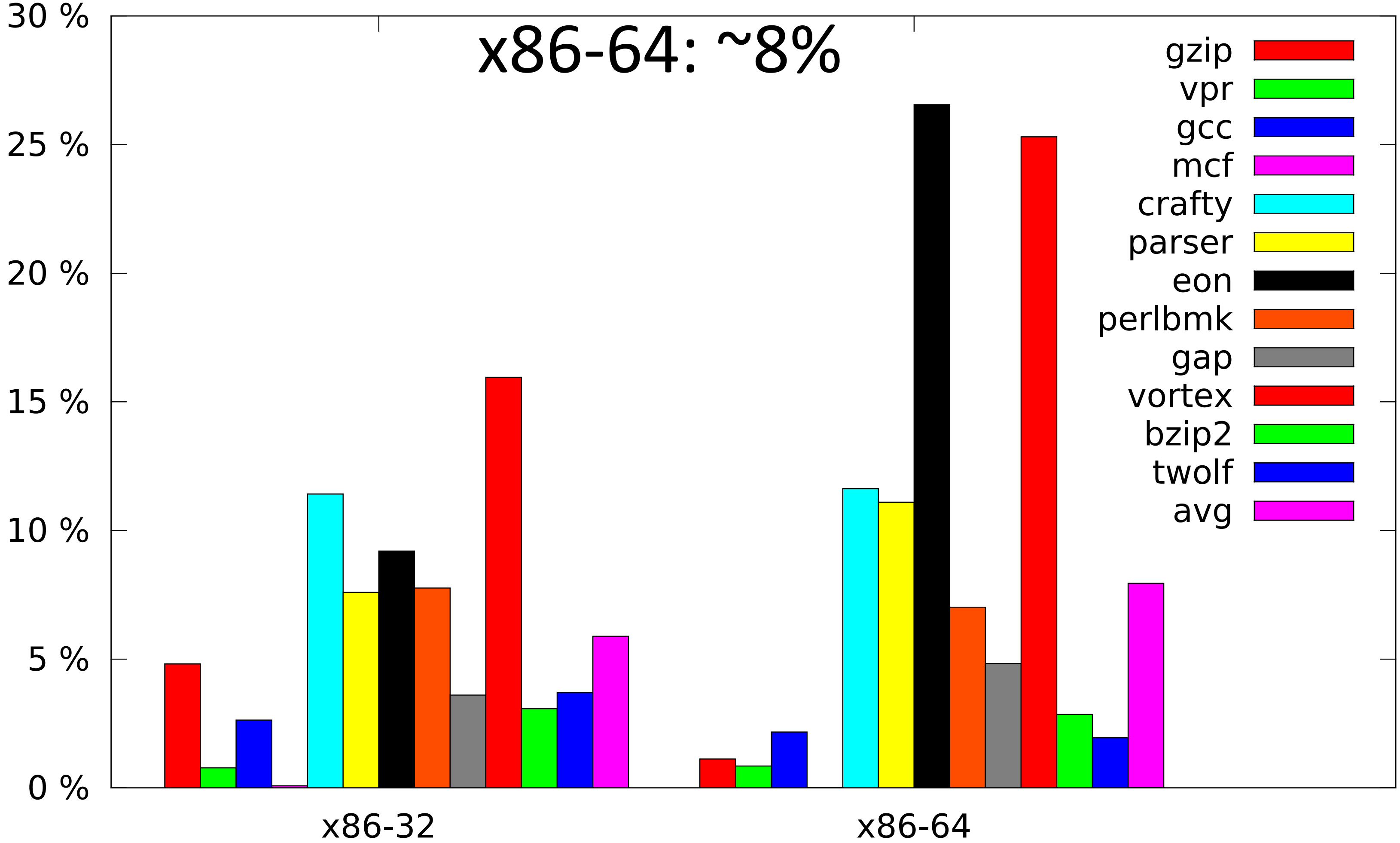
Performance Evaluation

- LLVM 2.9
- To demonstrate retargetability:
  - x86-32
  - x86-64 (small changes on x86-32)

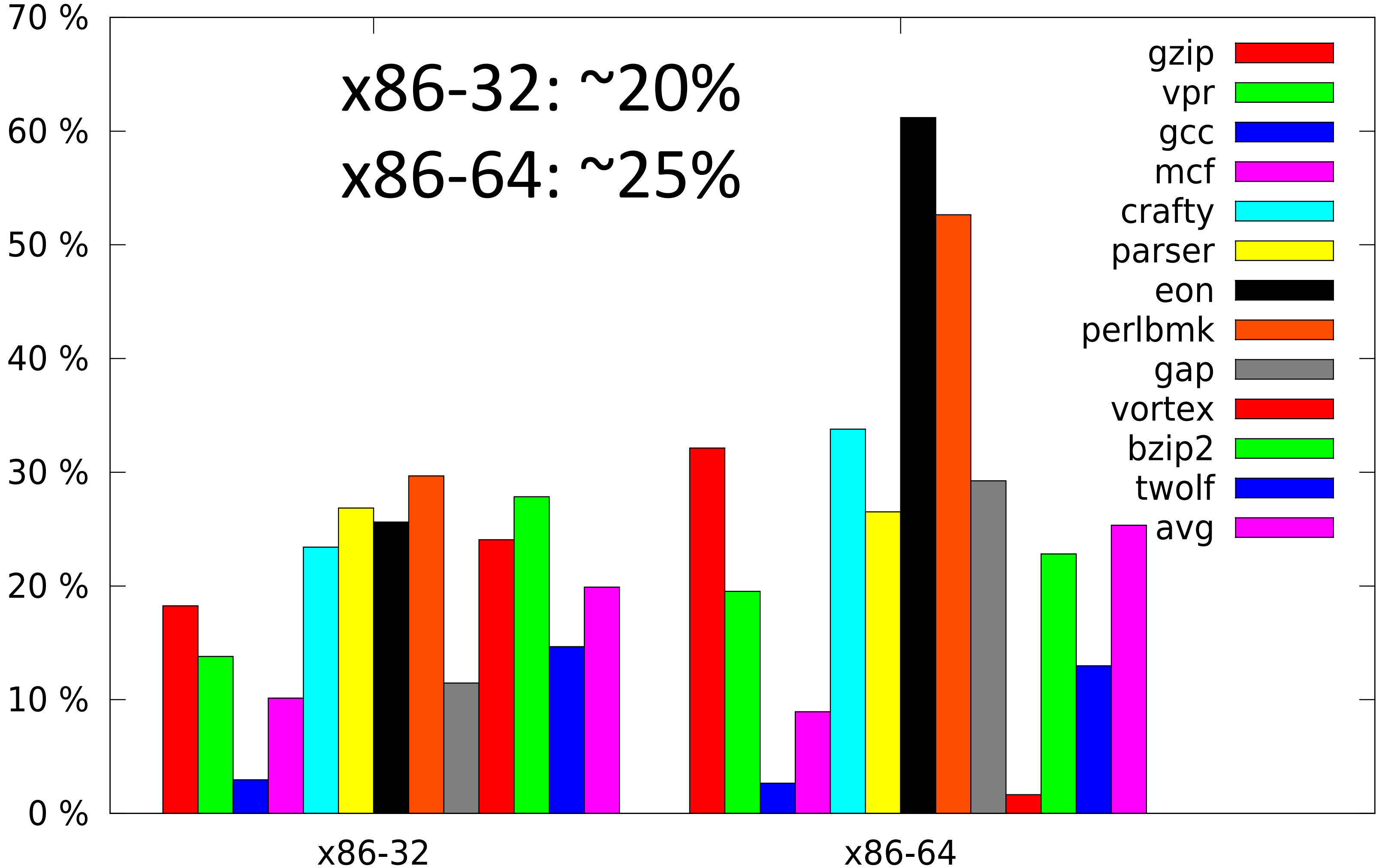
CFI Overhead on SPEC2k

x86-32: ~6%

x86-64: ~8%



Overhead of CFI with Data Sandboxing for Both Reads and Writes on SPEC2K



Compare with Previous work's performance

- Even though our framework is retargetable and trustworthy, the performance is competitive

Summary

- A retargetable framework for IRMs
- Optimizations on checks
  - Competitive performance
- Constraint language
- Range analysis based verifier

References

[CFI] Abadi et al. "Control-Flow Integrity – Principles, Implementations, and Applications", ACM CCS 2005

[csmith] Yang et al. "Finding and Understanding Bugs in C Compilers", PLDI 2011

[HyperSafe] Wang et al. "HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity", IEEE S&P 2010

[IRM] Erlingsson et al. "The Inlined Reference Monitor Approach to Security Policy Enforcement", doctoral dissertation, 2004

[Native Client] Yee et al. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code", IEEE S&P 2009

[Pittsfield] McCamant et al. "Evaluating SFI for a CISCI Architecture", USENIX Security 2006

[ROP] Roemer et al. "Return-oriented Programming: Systems, Languages, and Applications", ACM TISSEC 2012

[Shepherding] Kiriansky et al. "Secure Execution Via Program Shepherding", USENIX Security 2002

[SmashStack] Aleph One. "Smashing the stack for fun and profit", Phrack Magazine, 1996

[XFI] Erlingsson et al. "XFI: Software Guards for System Address Spaces", OSDI 2006

Strato: A Retargetable Framework for Low-Level Inlined-Reference Monitors

Thank you!

Questions?

Bin Zeng
zeb209@lehigh.edu

Gang Tan
gtan@cse.lehigh.edu

Úlfar Erlingsson
ulfar@google.com