Control-Flow Integrity For COTS Binaries

Mingwei Zhang and R. Sekar
Stony Brook University
USENIX Security 2013

Work supported in part by grants from AFOSR, NSF and ONR

Talk Outline

Motivation

Static analysis

Binary instrumentation

CFI properties and metric

Evaluation

Summary

Background

What is Control-Flow Integrity?

- **Program execution follows a statically-constructed control-flow graph (CFG)**

Why CFI?

- **a foundation for other low-level code defenses, e.g., SFI, sandboxing untrusted code, …**
- **defeats low level attacks on binaries**
  - **Code injection, ROP, JOP, …**
- **deterministic, not probabilistic defense**

Motivation for this work

- **Many previous works closely related to CFI**
  - CFI [Abadi et al 05, Abadi et al 2009, Zhang et al 2013]
  - Instruction bundling [MaCamant et al 2008, Yee et al 2009]
  - Indexed Hooks [2011], Control-flow locking [Bletsch et al 2011]
  - MoCFI [Davi et al 2012], Reins [Wartell et al 2012]…
- **Require compiler support, or binaries that contain relocation, symbol, or debug info**
- **Do not provide complete protection**
  - **Leave out executable, libraries, or the loader**
- **Have a difficult time balancing strength of protection and compatibility with large binaries**

Preview of Results

- **Robust on large and low-level binaries**
  - *glibc, gimp-2.6, adobe reader 9, firefox 5*
  - *executables as well as libraries*
- **Compatible yet strong policy**
  - *93% of ROP/JOP gadgets*
- **Good performance**
  - *~10% on CPU-intensive C/C++ benchmark (SPEC 2006), (~4% if restricted to C-programs)*
- **Limitations**
  - *Does not support obfuscated binaries or malware*
  - *No runtime code generation or JIT (yet)*
  - *Implemented for 32-bit Linux, tested with gcc and LLVM*

Key Challenges

- **Disassembly and Static analysis of COTS binaries**
- **Robust static binary instrumentation**
  - Without breaking low-level code
  - Transparency for position-independent code, C++ exceptions, etc.
- **Modular instrumentation**
  - Applied to executables and libraries
  - Enables sharing library code across many processes
- **Assess compatibility/strength tradeoff**

Disassembly Errors

- **Disassembly of non-code**
  - Tolerate these errors by leaving original code in place
- **Incorrect disassembly of legitimate code**
  - Instruction decoding errors (not a real challenge)
  - *Instruction boundary errors*
    - Harmful – our technique geared to find and repair them
  - Failure to disassemble (we avoid this)

Disassembly Algorithm

1 **Linear disassembly**

2 **Error detection**

- invalid opcode
- direct jump/call outside module address
- direct control into insn

3 **Error correction**

- Identify "gap:" data/padding disassembled as code
  - Scan backward to preceding unconditional jump
  - Scan forward to next direct or indirect target
    - *Indirect targets obtained from static analysis*

4 **Mark "gap," repeat until no more errors**

Static Analysis

Code pointers are needed:

- **to correct disassembly errors**
- **to constrain indirect control flow (ICF) targets**

We classify code pointers into categories:

- **Code Pointer Constants (CK)**
- **Computed Code Pointers (CC)**
- Exception handlers (EH)
- Exported symbols (ES)
- Return addresses (RA)

Static Analysis

- **Code pointer constants**
  - *Scan for constants :*
    - *at any bye offset within code and data segments*
    - *fall within the current module*
    - *point to a valid instruction boundary*
- **Computed code pointers**
  - *Does not support arbitrary arithmetic, but targets jump tables*
  - *Uses static analysis of code within a fixed-size window preceding indirect jump*

Talk Outline

Motivation

Static analysis

Binary instrumentation

CFI properties and metric

Evaluation

Summary

Instrumented Module

ELF header phdr	Original code metadata, .rodata	Original data ..bss	New code New data

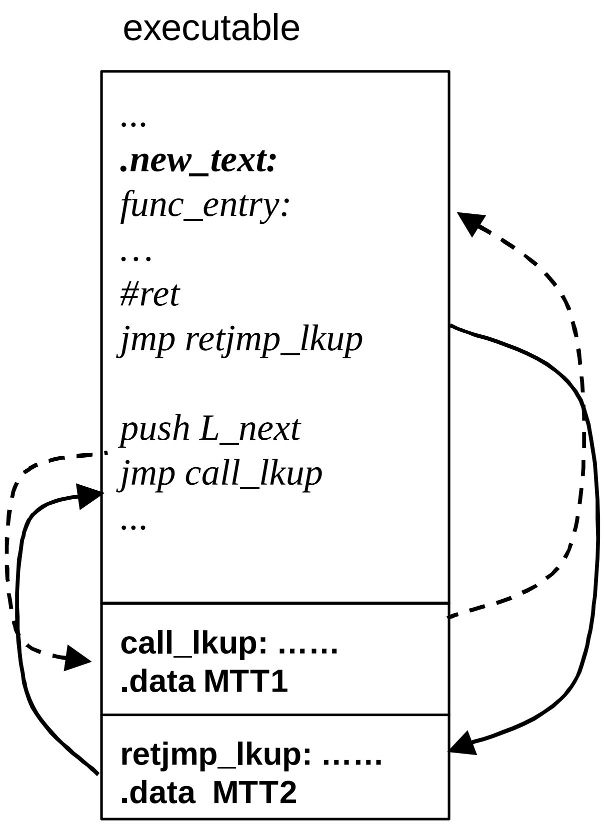
- **Translating function pointers**
  - *Appear as constants in code, but can't statically translate*
  - *Solution (from DBT): Runtime address translation*
- **Full transparency:** all code pointers, incl. dynamically generated ones, target original code [Bruening 2004]
  - *Important for supporting unusual uses of code pointers*
    - *To compute data addresses (PIC-code , data embedded in code)*
    - *C++ exception handling*

Static Instrumentation for CFI

- **Goal: constrain branch targets to those determined by static analysis**
  - *Direct branches: nothing to be done*
  - *Indirect branches: check against a table of (statically computed) valid targets*
- **Key observation**
  - *CFI enforcement can be combined with address translation!*

Modularity

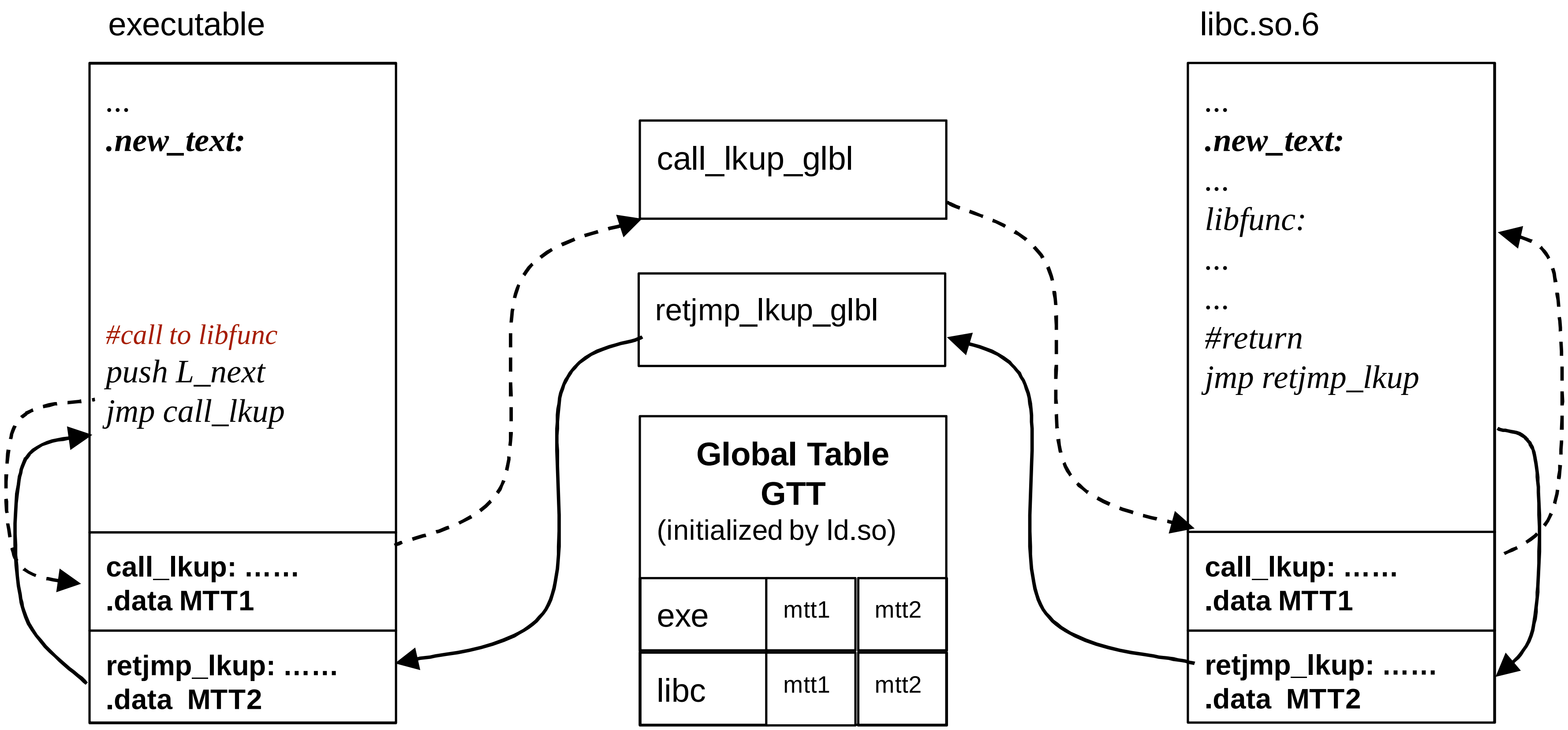
Intra-module control transfer: MTT



What if the target is out side of the module ?

Modularity

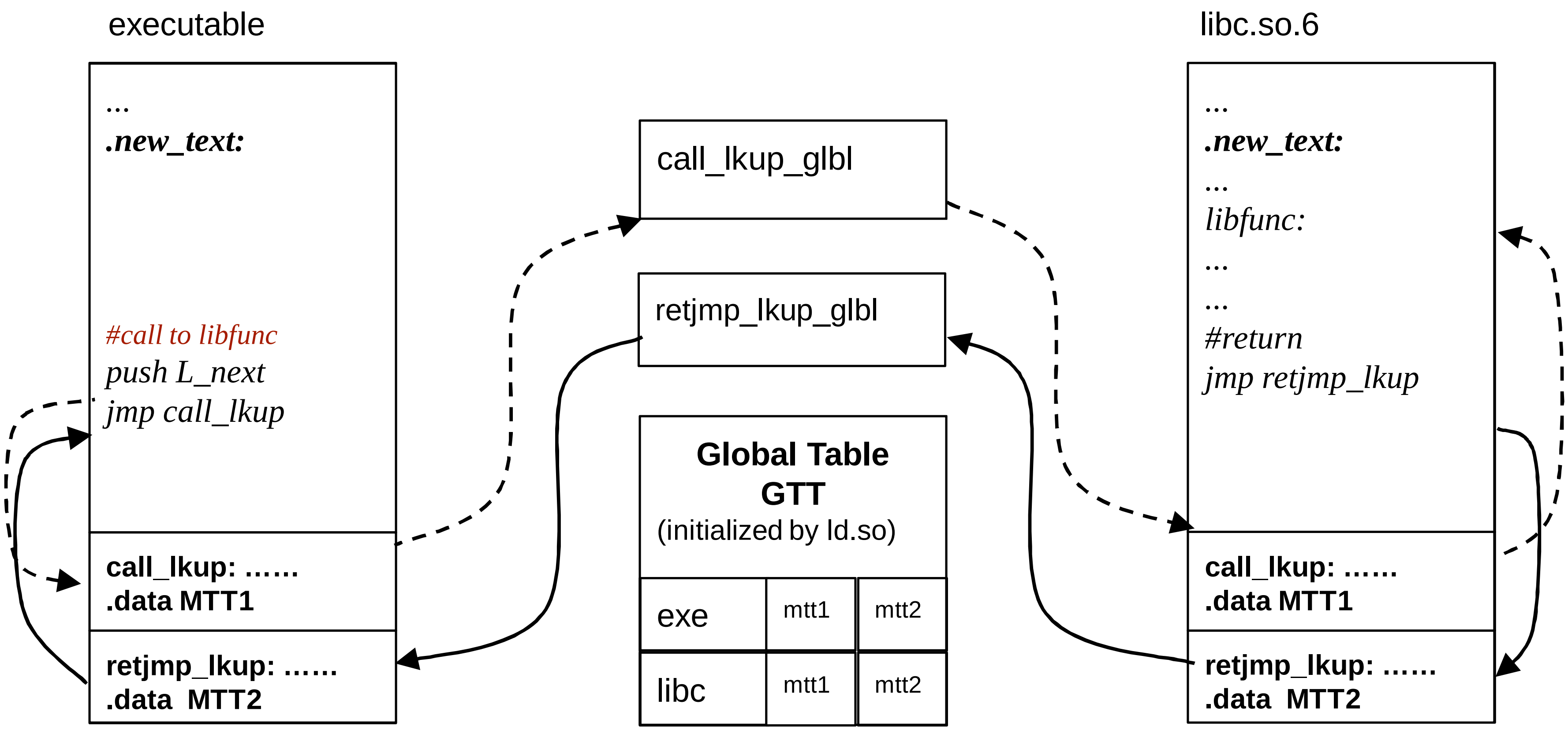
Inter-module control transfer: GTT



update of GTT is done in ld.so

Modularity

Code injection: null GTT entry



GTT only maps code !

Talk Outline

Motivation
Static analysis
Binary instrumentation
CFI properties and metric
Evaluation
Summary

Basic version of CFI

- **return: target next of call**
- **call/jmp: target any function whose address is taken**
  - Obtainable from relocation info ("reloc-CFI")
  - matches implementation described in [Abadi et al 2005]

- **How to cope with missing relocation info?**
  - *Use static analysis to over-approximate function addresses taken*
- **"Strict-CFI"**

CFI Real-World Exceptions

- **special returns**
  a. *as indirect jumps (lazy binding in ld.so)*
  b. *going to function entries (setcontext(2))*
  c. *not going just after call (C++ exception)*
- **calls used to get PC address**
- **jump as a replacement of return**

binCFI Policy

bin-CFI	Returns (RET), Indirect Jumps (IJ)	Indirect Calls (IC), PLT jumps (PLT)
Return addresses (RA)	Y	
Exception handling addresses (EH)	Y (C++)	
Exported symbol addresses (ES)		Y
Code pointer constants (CK)	Y (C++, Context switch)	Y (GNU_IFUNC)
Computed code addresses (CC)	Y (return as jump)	Y (GNU_IFUNC)

Well, is this policy too weak ?

Measuring “Protection Strength”

- **Average Indirect target Reduction (AIR)**
  a. *T_j: number of possible targets of jth ICF branch*
  b. *S: all possible target addresses (size of binary)*

$$\frac{1}{n}\sum_{j=1}^{n}\left(1-\frac{|T_j|}{S}\right)$$

- **AIR is a general metric that can be applied to other control-flow containment approaches**

Coarser versions of CFI

bundle-CFI:

- **all ICF targets aligned on 2^n-byte boundary, n = 4 (PittSFIeld) or 5 (Native Client)**

instr-CFI: the most basic CFI

- **all ICFTs target instruction boundaries**

AIR metric (single module)

Name	Reloc CFI	Strict CFI	**Bin CFI**	Bundle CFI	Instr CFI
perlbench	98.49%	98.44%	**97.89%**	95.41%	67.33%
bzip2	99.55%	99.49%	**99.37%**	95.65%	78.59%
gcc	98.73%	98.71%	**98.34%**	95.86%	80.63%
gobmk	99.40%	99.40%	**99.20%**	97.75%	89.08%
……	……	……	**……**	……	……
average	**99.13%**	**99.08%**	**98.86%**	**96.04%**	**79.27%**

- **Loss due to use of static analysis is negligible**
- **Loss due to binCFI relaxation is very small**

Evaluation

Disassembly testing

Real world program testing

Gadget elimination

Disassembly Testing

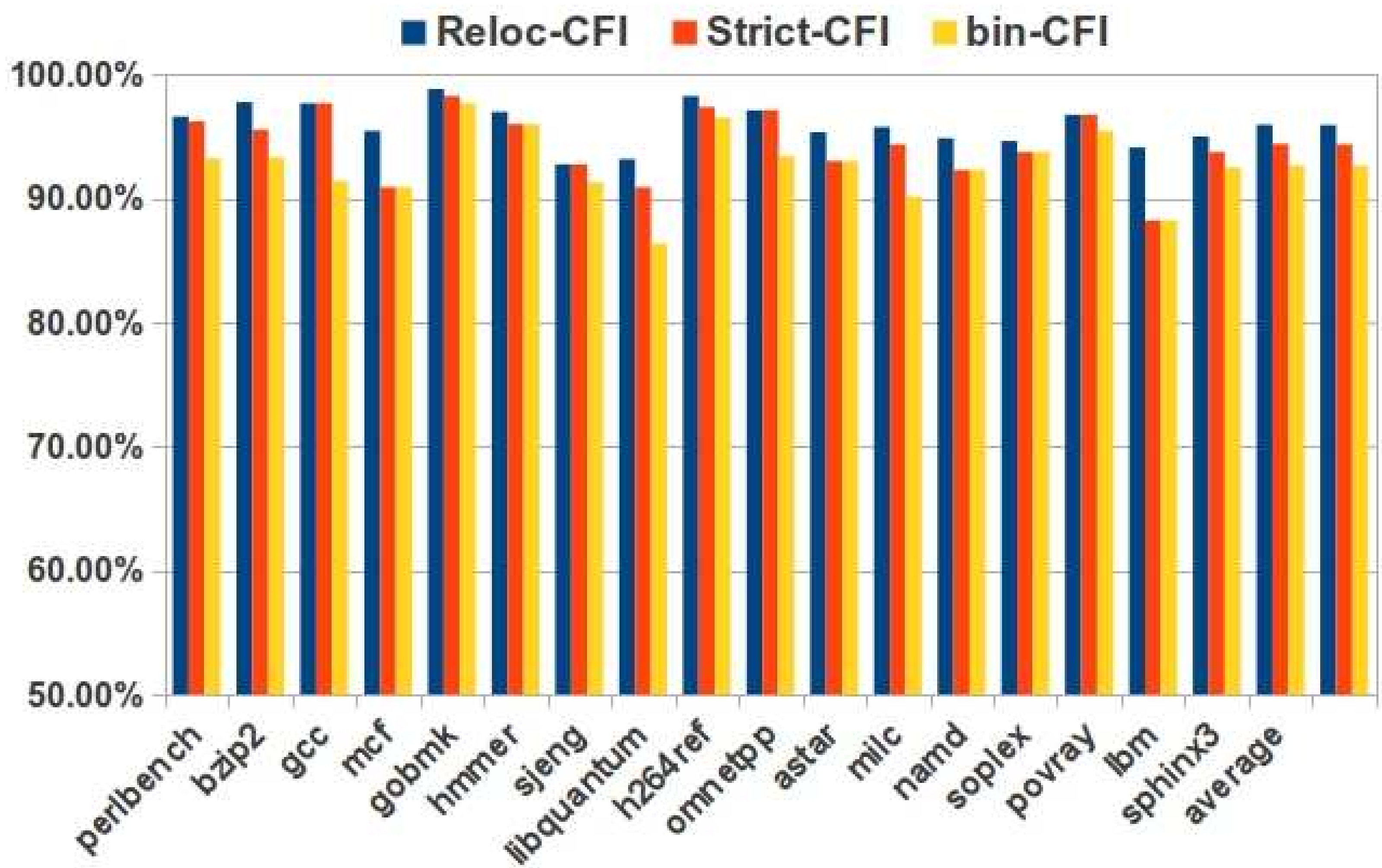
Module	Package	Size	Instruction#	errors
libxul.so	firefox-5.0	26M	4.3M	0
gimp-console-2.6	gimp-2.6.5	7.7M	385K	0
libc.so	glibc-2.13	8.1M	301K	0
libnss3.so	firefox-5.0	4.1M	235K	0
……	……	……	……	……
Total		58M	5.84M	0

"diff" compiler generated assembly and our disassembly

Real world program testing

Application Name	Experiment
firefox 5 (no JIT)	open web pages
acroread9	open 20 pdf files; scroll;print;zoom in/out
gimp-2.6	load jpg picture, crop, blur, sharpen, etc.
Wireshark v1.6.2	capture packets on LAN for 20 minutes
lyx v2.0.0	open a large report; edit; convert to pdf/dvi/ps
mplayer 4.6.1	play an mp3 file
…...	….........
Total:	12 real world programs

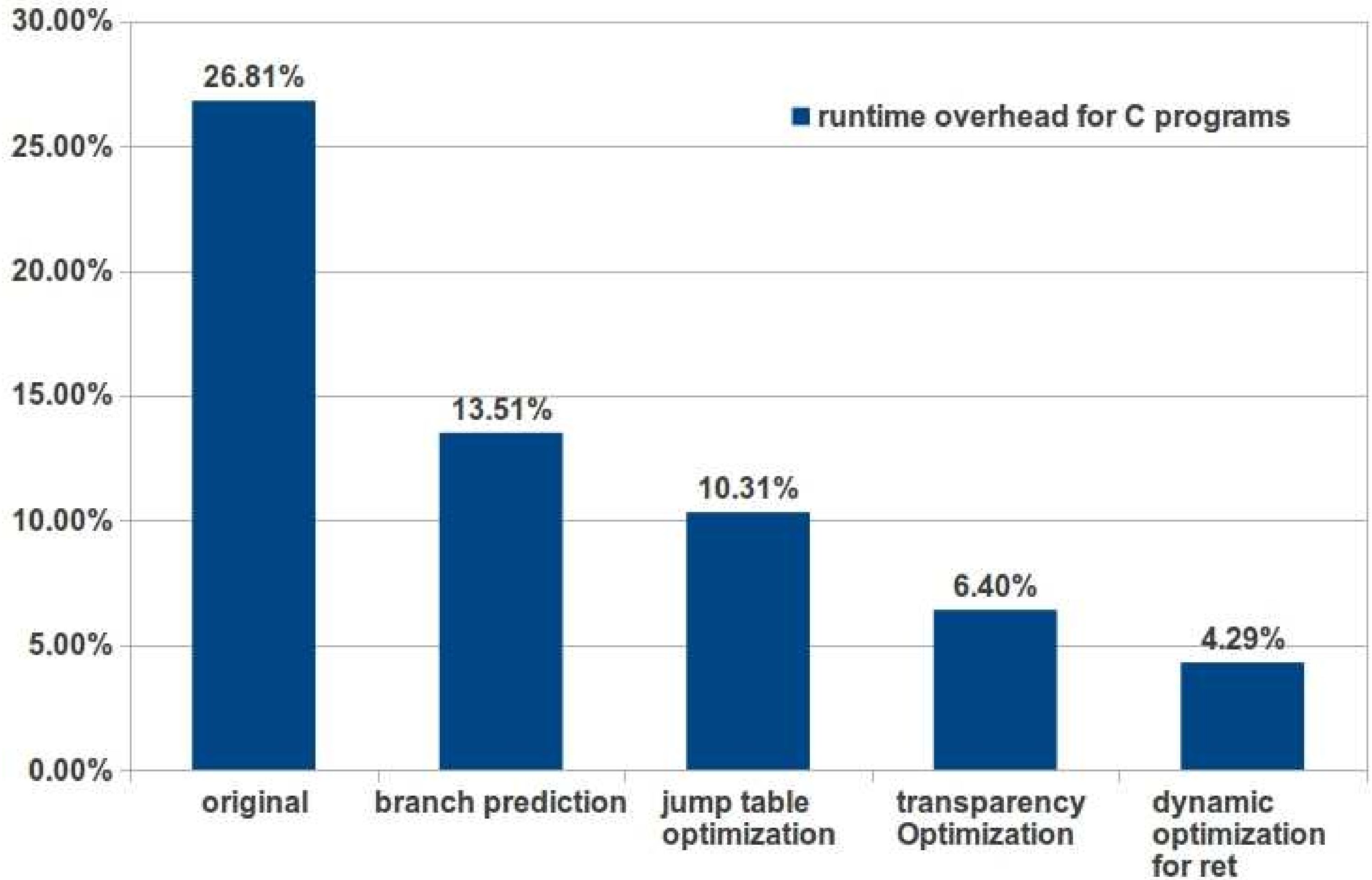
Gadget Elimination



Optimizations

- Branch prediction: Optimized translation of calls and returns, avoiding indirect jumps
- Jump table: Avoid runtime address translation in jump tables
- Transparency optimization: Avoid address translation for returns (but check validity)
- Dynamic optimization for returns: Fast check for most frequent target

Effect of Optimizations



Questions?