Practical Control Flow Integrity & Randomization for Binary Executables

Chao Zhang
Peking University

Tao Wei
Peking University
UC Berkeley

Zhaofeng Chen
Peking University

Lei Duan
Peking University

László Szekeres
Stony Brook University

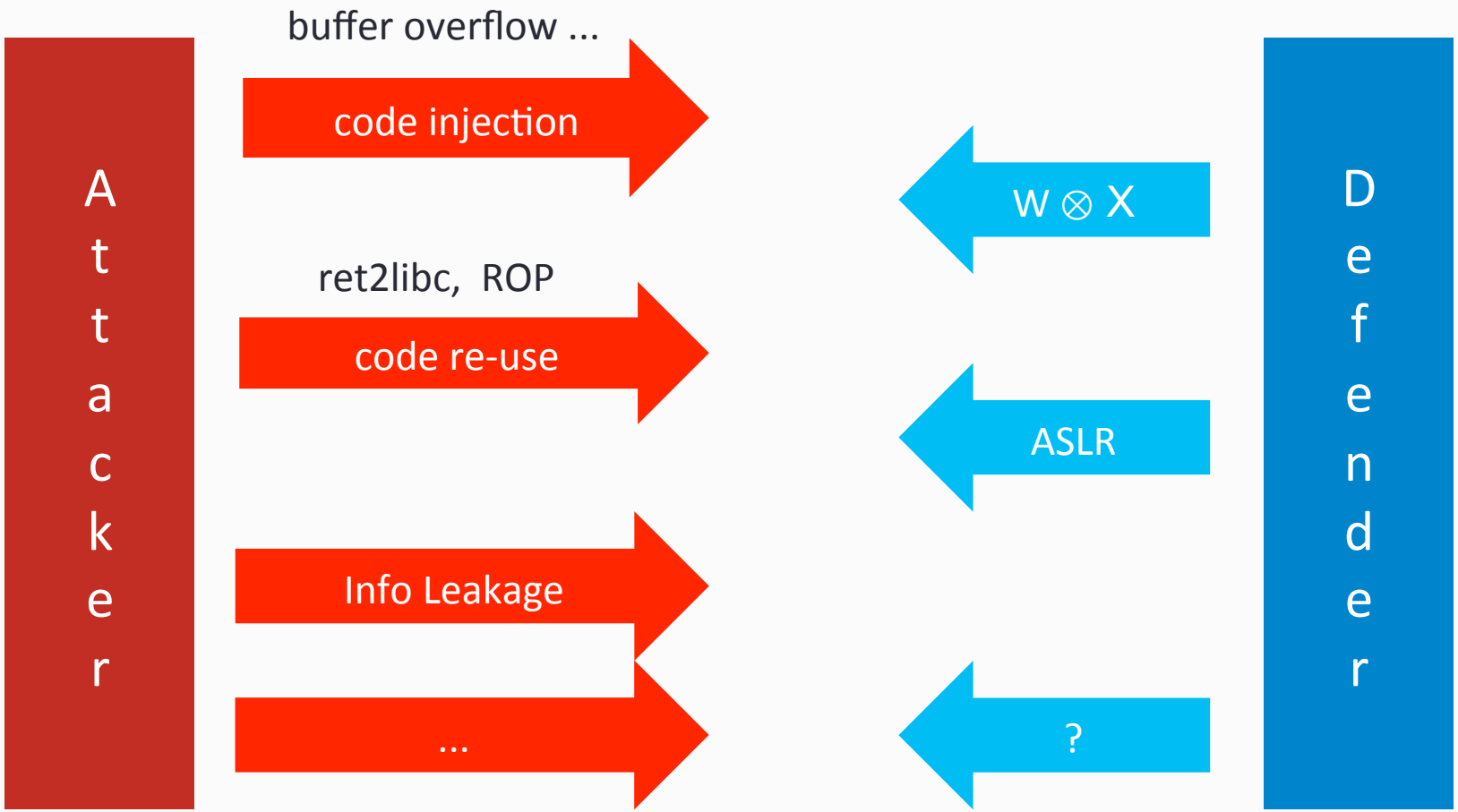
Stephen McCamant
University of Minnesota

Dawn Song
UC Berkeley

Wei Zou
Peking University

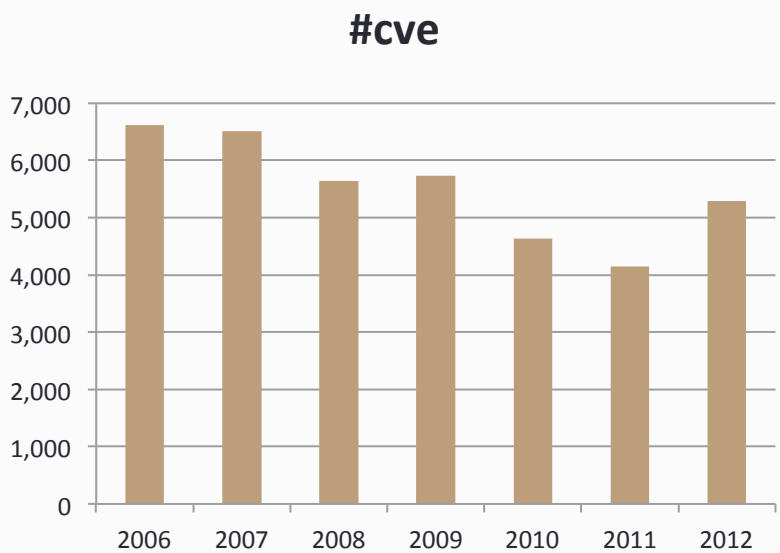
PEKING UNIVERSITY · Berkeley UNIVERSITY OF CALIFORNIA · Stony Brook University · UNIVERSITY OF MINNESOTA Driven to Discover℠

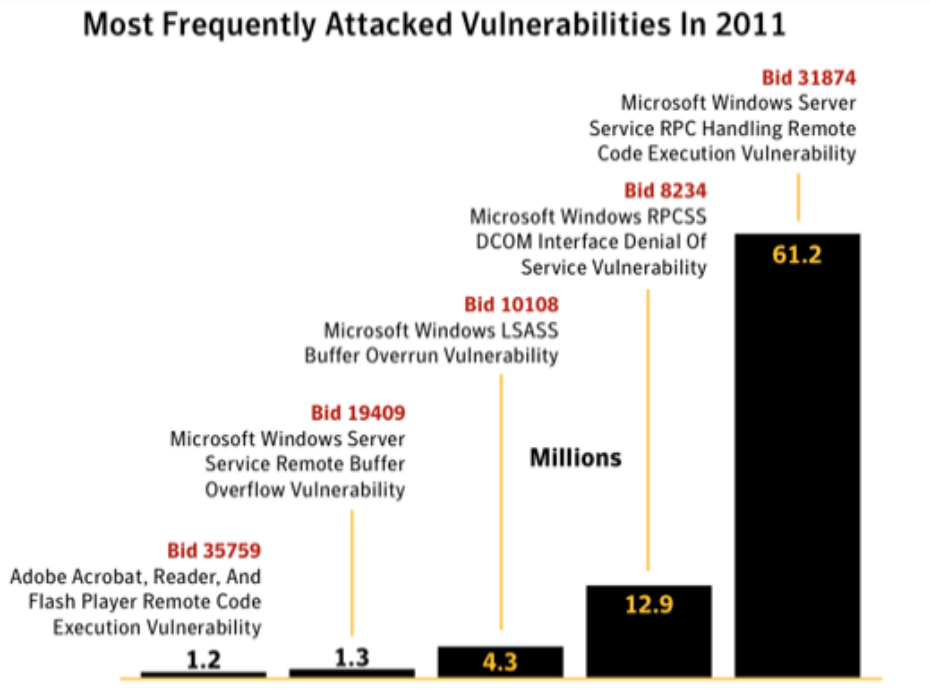
Eternal War in Memory (*Oakland'13*)



Reactive Defense is not adequate

- > 5,000 CVE vulnerabilities each year.
  - Many more are under the iceberg.
- Vulnerabilities are exploited millions of times.





Symantec Internet Security Threat Report

Proactive Defense: Control Flow Integrity

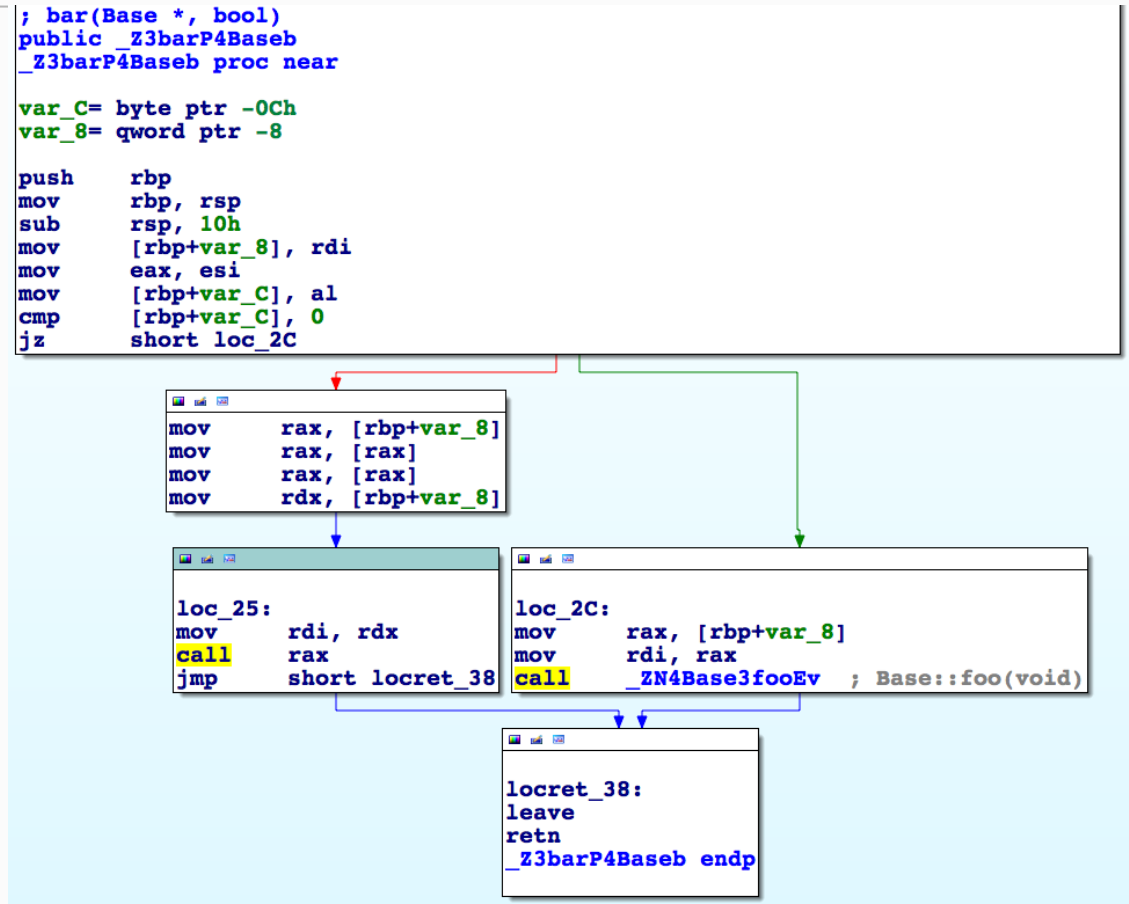
- Control Flow Integrity, CFI [CCS'05]
- Security policy
  - Each run-time control transfer must comply with programmer's compile-time intent, i.e., the control-flow graph
- A strong guarantee that can be reasoned about formally
  - ROP, JOP, UAF
- A foundation for other low-level code defenses
  - SFI, sandboxing untrusted code
- Deterministic, not probabilistic defense

Control-Flow Graph

```
class Base{
public:
        virtual void vfoo();
                void  foo();
};

class Derived: public Base{
public:
        virtual void vfoo();
                void  foo();
};

void bar(Base* b, bool virt)
{
        if(virt)
                b->vfoo();
        else
                b->foo();
}
```

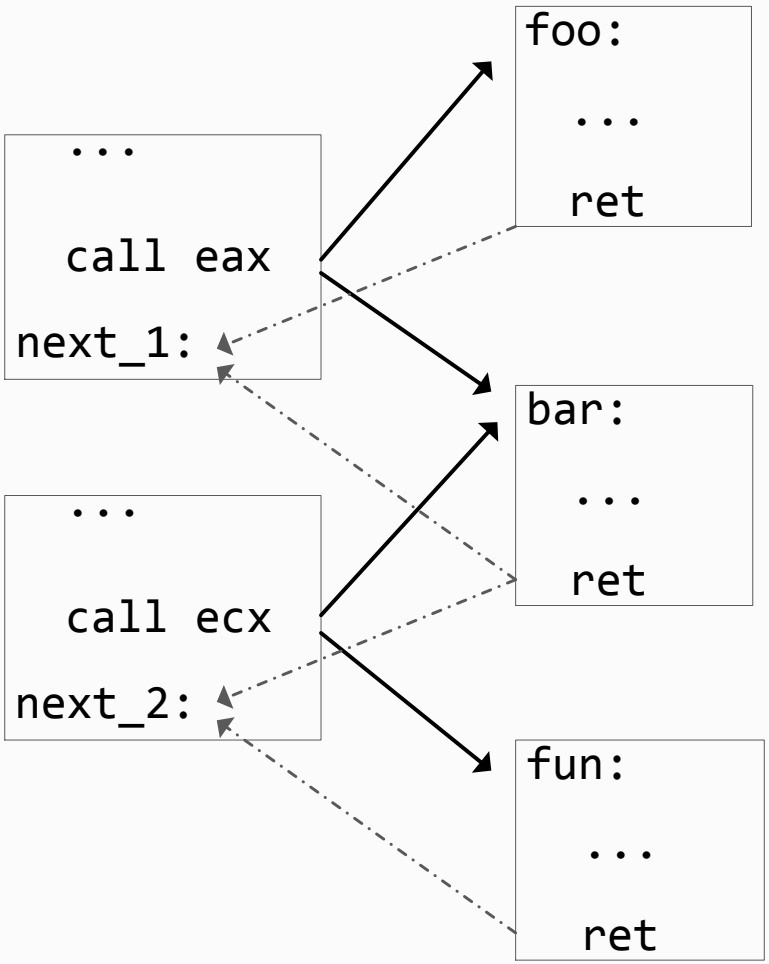


- conditional jump (jz …): target relative address
- direct jump/call: target relative or absolute address
- indirect jump/call: target? (read or computed from memory)
- return: target return address on the stack

Scope of Control-Flow Integrity Protection

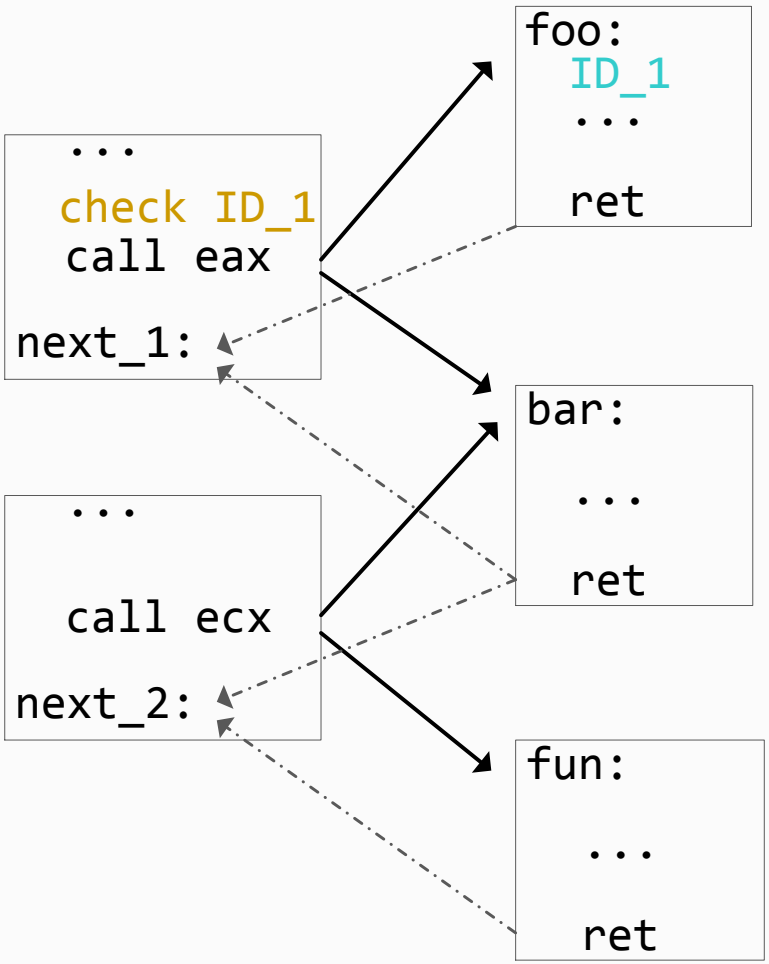
Control Transfer Method	Transfer Targets	Integrity Protection
Exceptions	Exception handlers	SafeSEH, SEHOP
Direct CALL/JMP, Conditional jump (JZ...)	Hard-coded function pointers (embedded in the code)	W⊗X
Indirect CALL/JMP	In-memory function pointers	**CFI**
RET instructions	On-stack return addresses	/GS, shadow stack, **CFI**

Control Flow Integrity (CCS'05)



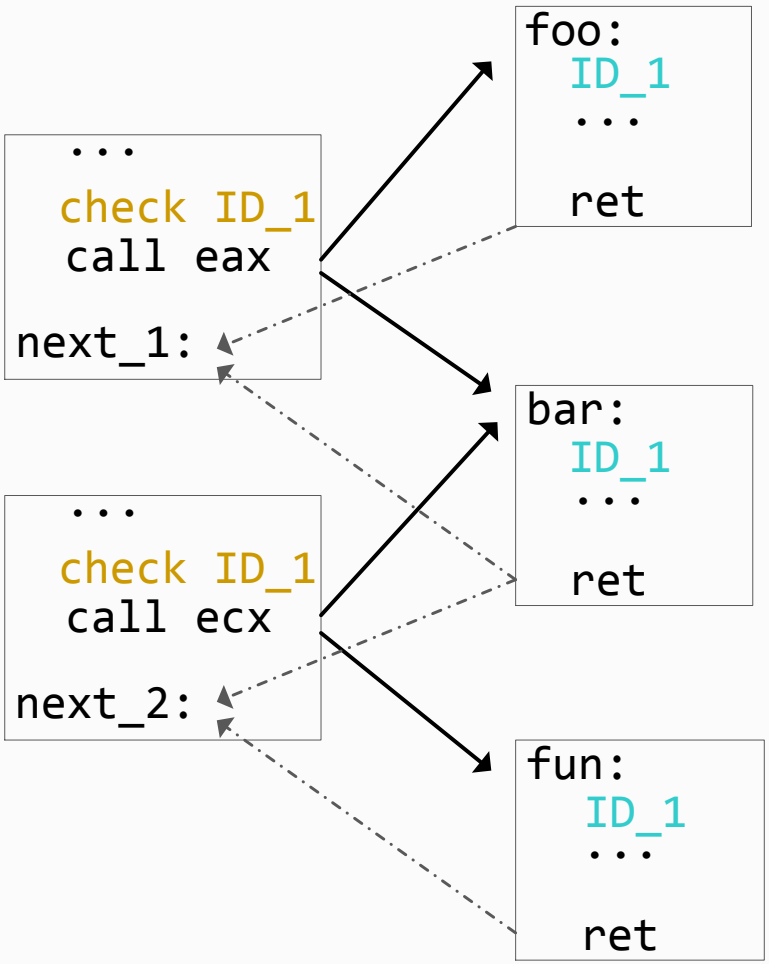
- build Control Flow Graph
  - source code/debug info
  - all modules

Control Flow Integrity (CCS'05)



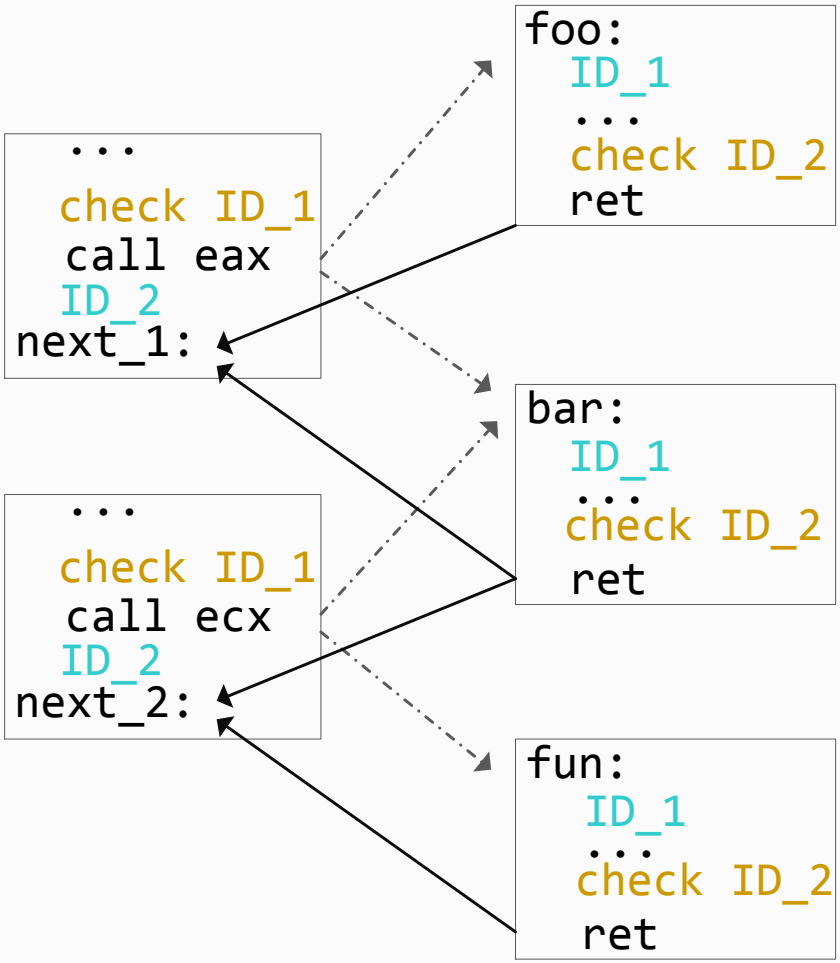
- build Control Flow Graph
- binary rewriting
  - instrument IDs before transfer targets
  - validate IDs before control transferring

Control Flow Integrity (CCS'05)



- build Control Flow Graph
- binary rewriting
- diversity of IDs

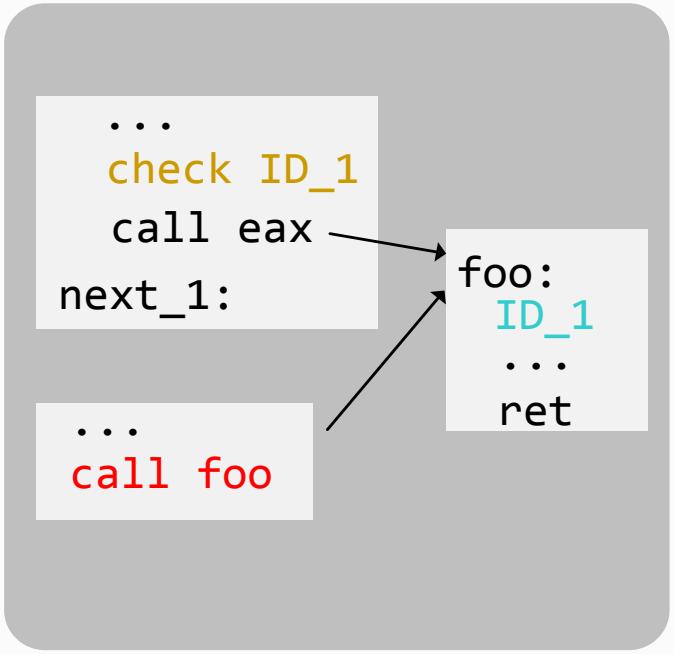
Control Flow Integrity (CCS'05)



- build Control Flow Graph
- binary rewriting
- diversity of IDs
- all indirect transfers
  - indirect call
  - indirect jmp
  - ret

Challenges faced by CFI

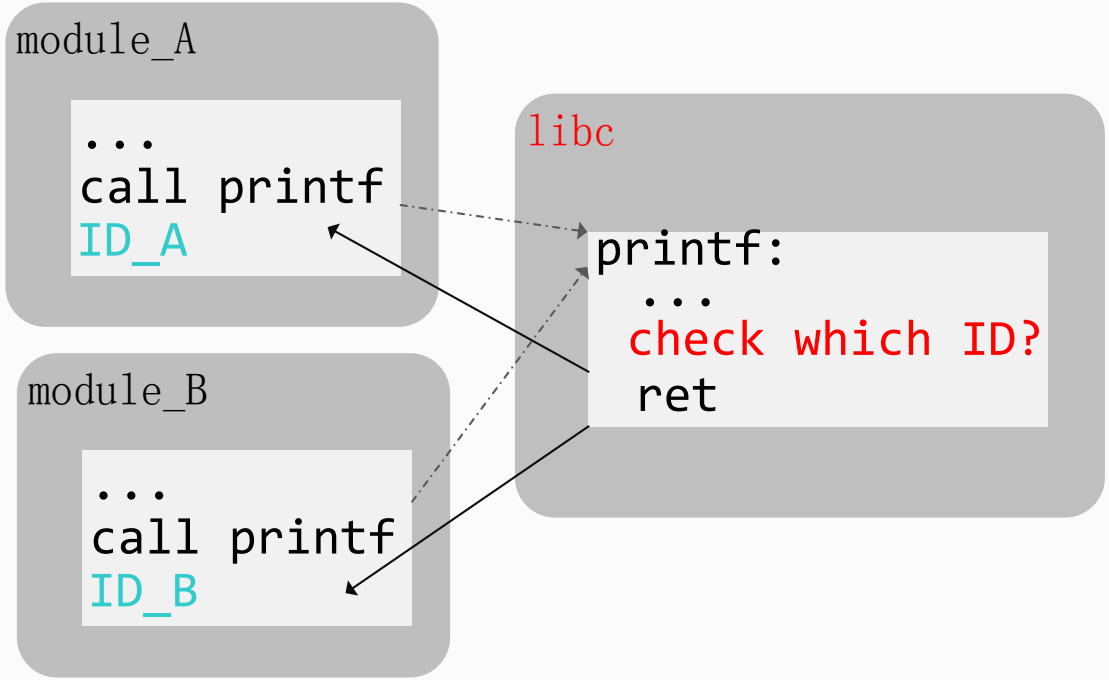
- performance overhead: average 15%, max 46%



- ID is embedded in a slow *prefetchnta* instruction
  - executed each time the transfer target is reached
- introduce overhead even if *foo* is called/jumped to directly
  - ✓ most calls/jumps are direct

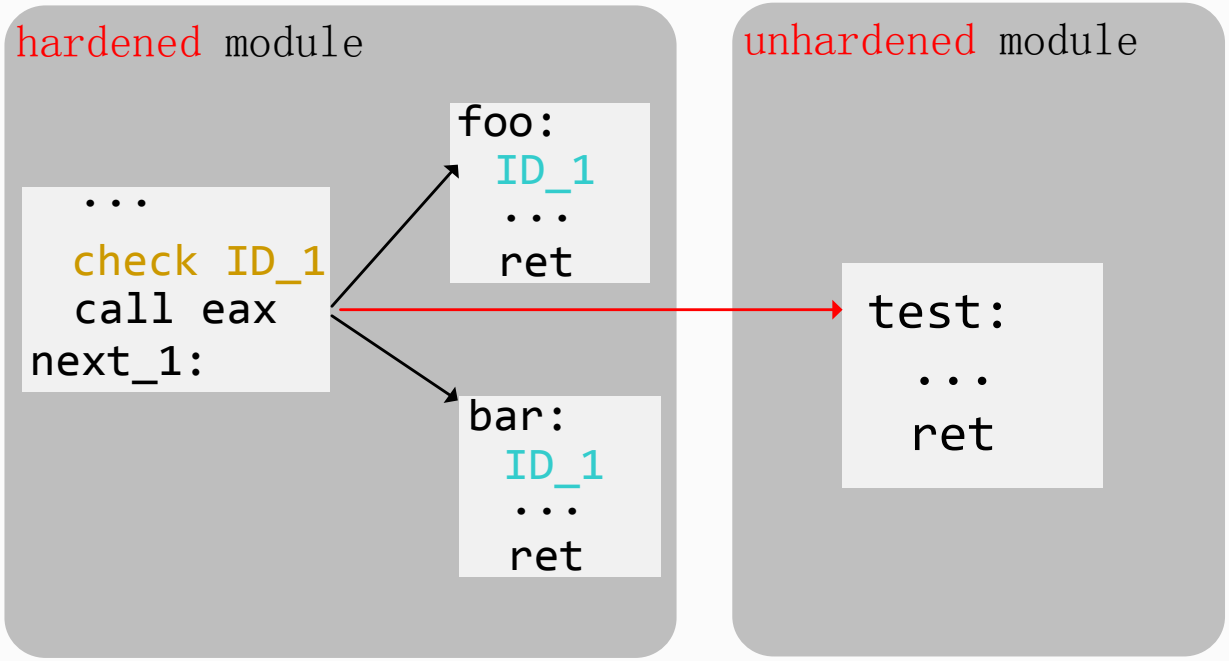
Challenges faced by CFI

- performance overhead
- modularity support
  - A single module cannot be hardened independently



Challenges faced by CFI

- performance overhead
- modularity support
- backward compatibility



Challenges faced by CFI

- performance overhead
- modularity support
- backward compatibility
- source code/debug info dependency
  - to build Control Flow Graph

Motivation

- A light-weight CFI solution with a strong protection
  - performance overhead ✓ ✓ ✓
  - modularity support ✓ ✓ ✓
  - backward compatibility ✓ ✓ ✓
  - source code independent ✓ ✓ ✓

Assumption

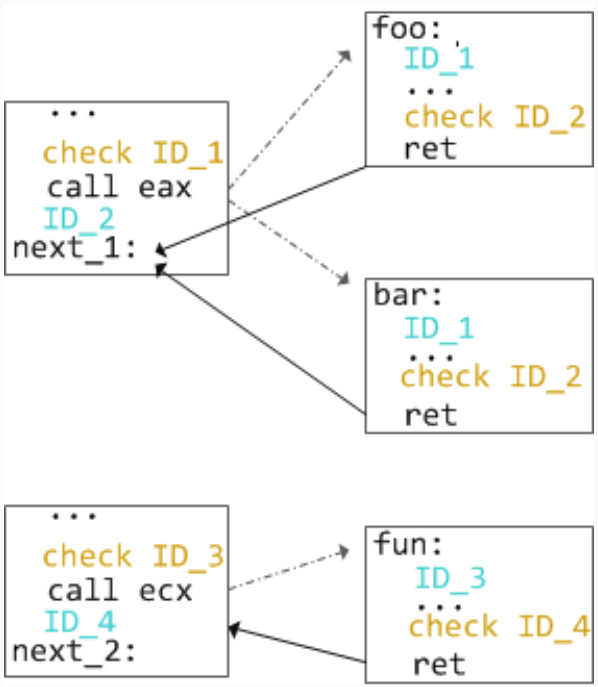
- ASLR and W⊗X (e.g. DEP) are deployed
- NO self-modifying code or dynamically generated code.
- Limited information disclosure vulnerabilities
  - e.g., cannot read entire memory regions

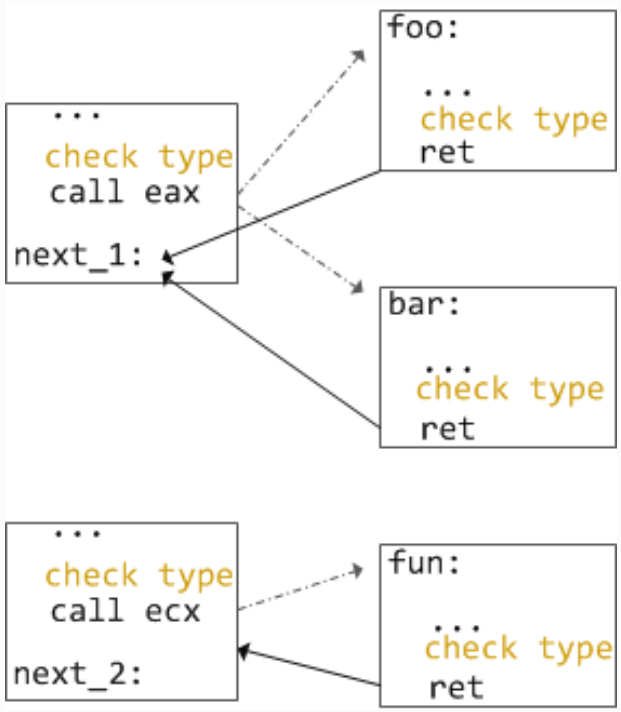
Outline

- Motivation
- Intuition & Design
- Implementation
- Security Enhancement
- Evaluation
- Conclusion

Intuition

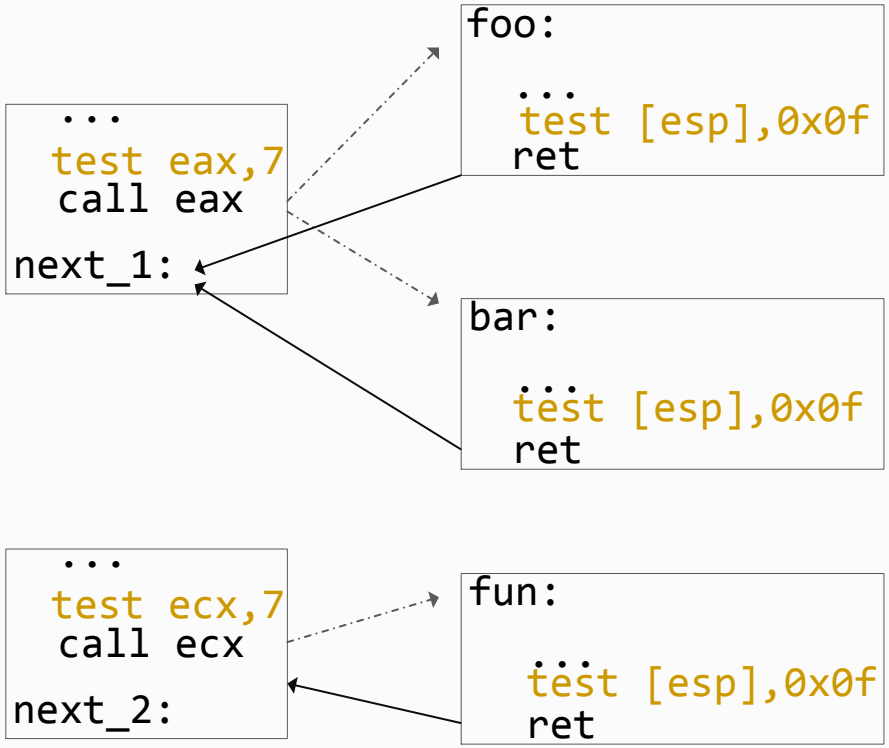
- Checking transfer targets' kind, rather than IDs
  - indirect call/jmp can only transfer to function-pointer stubs
  - returns can only transfer to return-address stubs





Intuition: efficient check

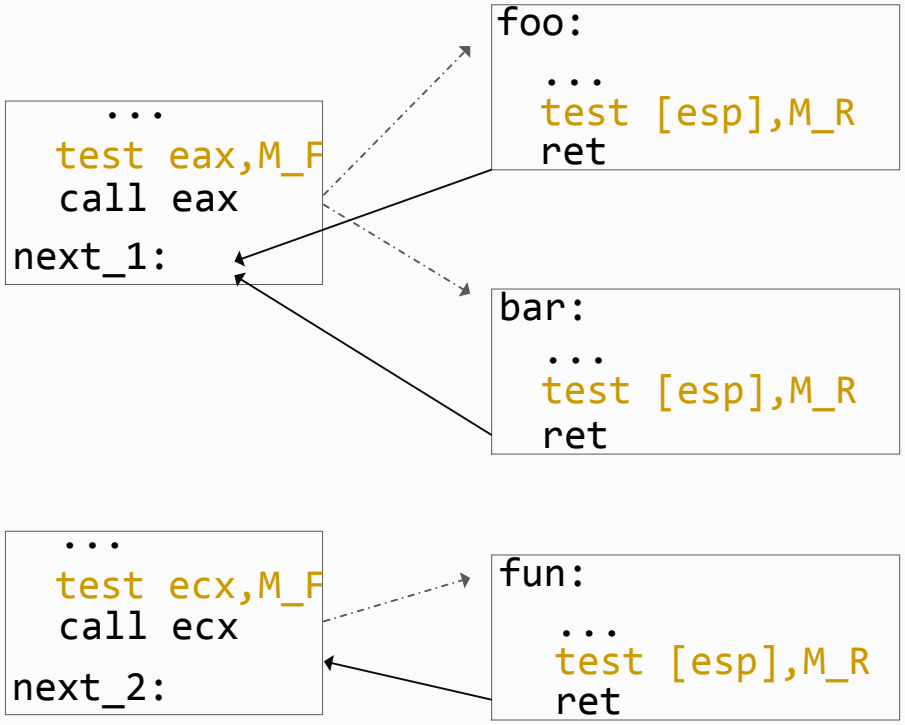
- memory alignment
  - function pointer stubs are 8-bytes aligned
  - return address stubs are 16-bytes aligned



- fast bit testing
  - performance overhead ✓
- No IDs
  - performance overhead ✓ ✓
- only check type
  - modularity support ✓ ✓ ✓

Intuition: security

- Transfer targets must be within memory region Springboard
  - 27th bit is 0



M_F = 0x8000007

M_R = 0x800000f

- Policy: all indirect transfer targets must be aligned stubs in Springboard.

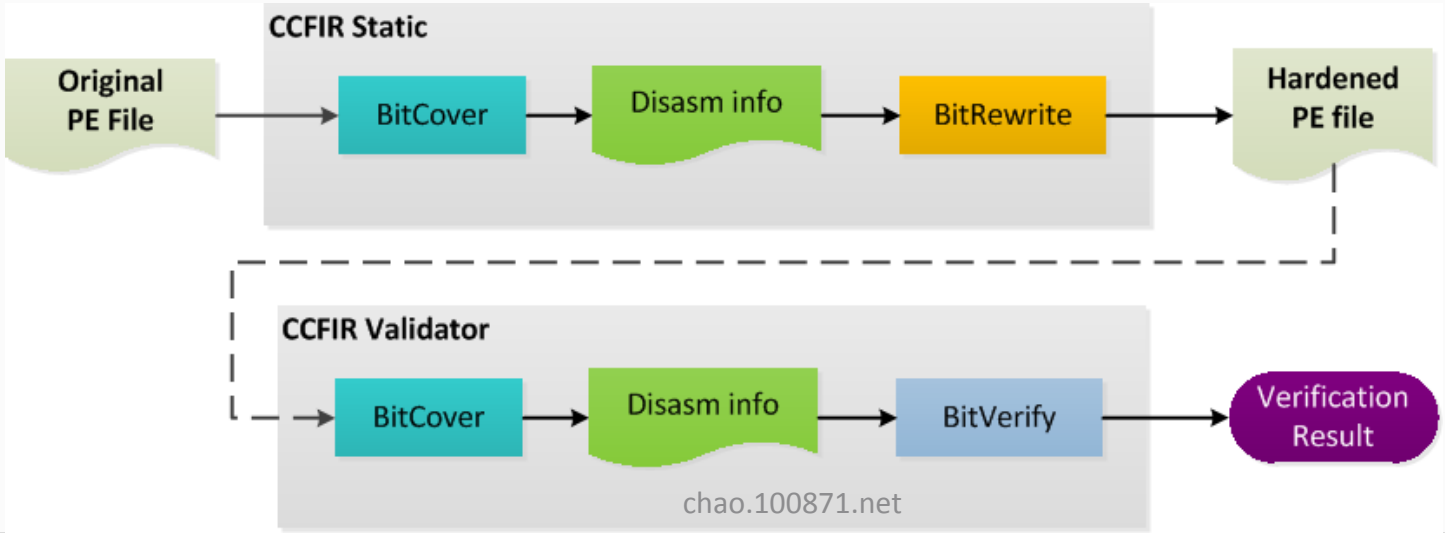
Design: Memory layout with Springboard

Executable ?	Bits				Meaning
	27	26	3	2-0	
No	*	*	*	***	Non-executable section
Yes	1	*	*	***	Normal code section
Yes	0	*	*	!000	Invalid entry in Springboard
Yes	0	*	1	000	fp-stub entry in Springboard
Yes	0	1	0	000	Sensitive ret-stub entry in Springboard
Yes	0	0	0	000	Normal ret-stub entry in Springboard

- Springboard vs. normal code section (27th bit)
- Entries in the Springboard are all aligned. (0-2 bits)
- Three kinds of stubs in Springboard
  - fp-stub entries vs. ret-stub entries (3rd bit)
  - sensitive vs. normal ret-stub (26th bit)
    - normal ret-stubs cannot return into the middle of sensitive functions
- One or two fast bit testing instructions are sufficient

Architecture of our solution: CCFIR

- Find out all indirect transfer instructions and their targets
  - BitCover: disassemble target binary
- Redirect transfer targets to aligned stubs in Springboard
- Validate transfer targets
  - BitRewrite: binary rewriting
- Verify the hardened binary
  - BitVerify



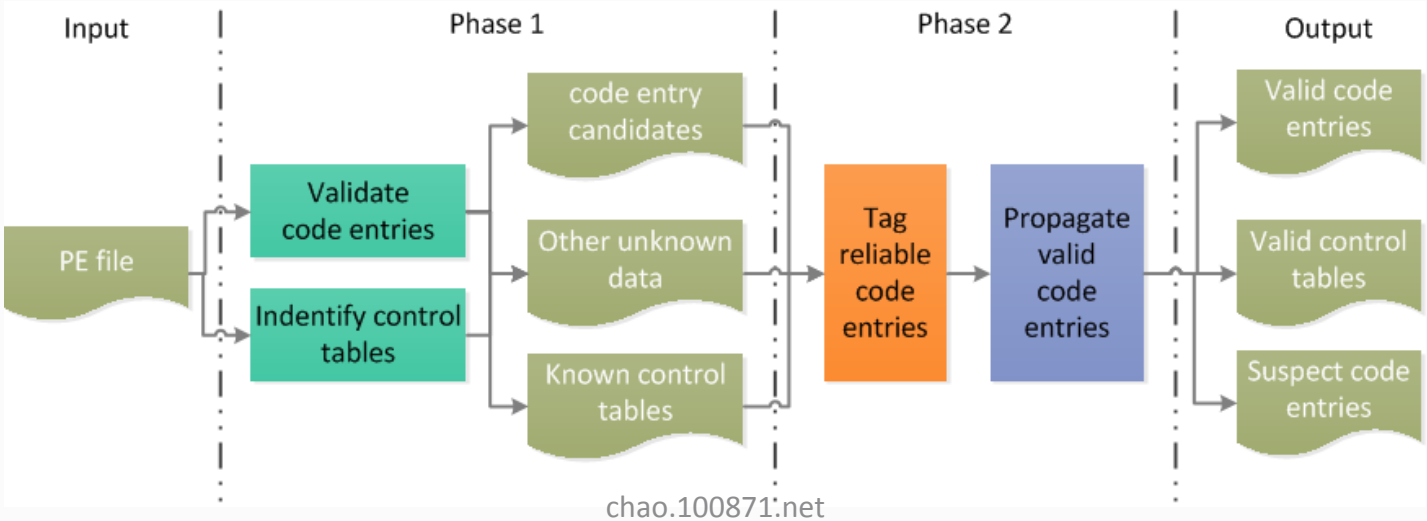
Outline

- Motivation
- Intuition & Design
- Implementation
- Security Enhancement
- Evaluation
- Conclusion

BitCover: disassembling

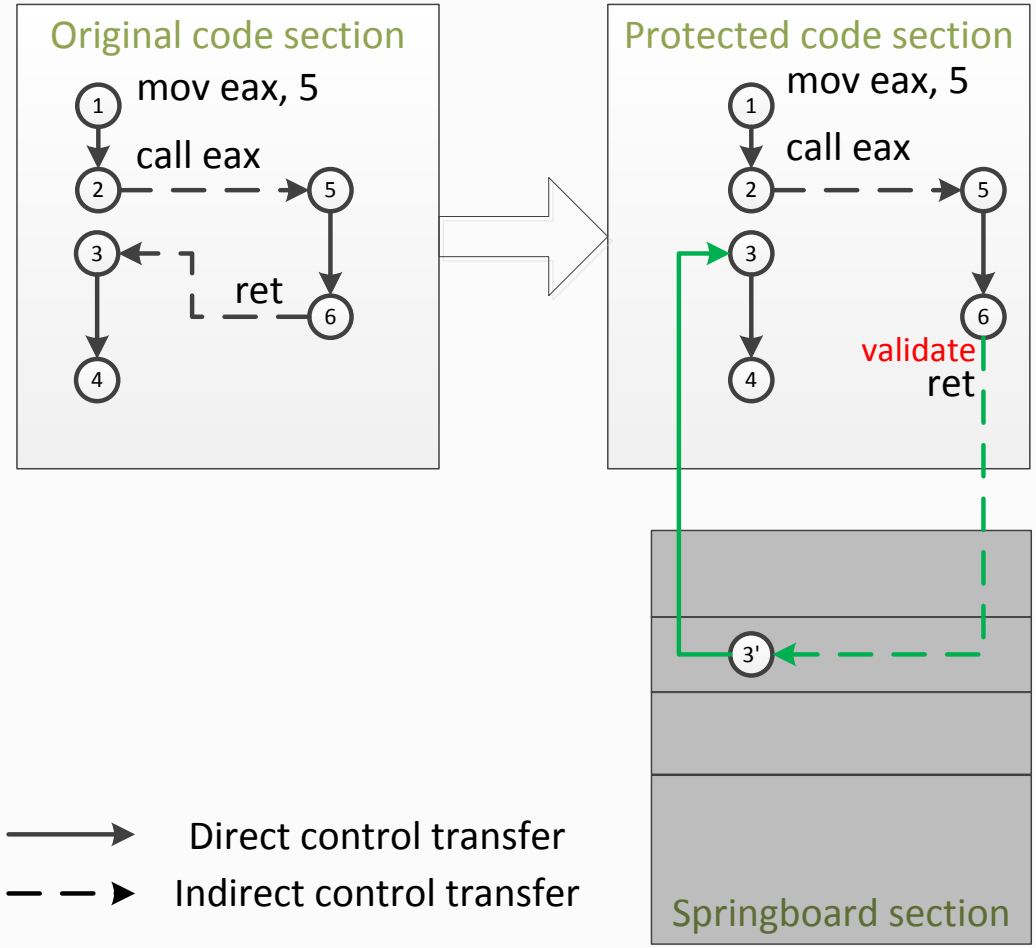
- performance overhead ✓ ✓
- modularity support ✓ ✓ ✓
- backward compatibility
- source code independent ✓ ✓ ✓

- Target: ***normal*** binaries generated by modern compilers
  - Support ASLR
    - On Windows, it implies the executable contains relocation information
    - We can identify all legal function pointers (indirect call/jmps' targets)
- Automatically disassemble most of ***normal*** binaries
  - Based on well-defined rules
  - More accurate than IDA Pro, especially designed for binary rewriting



BitRewrite: binary rewriting

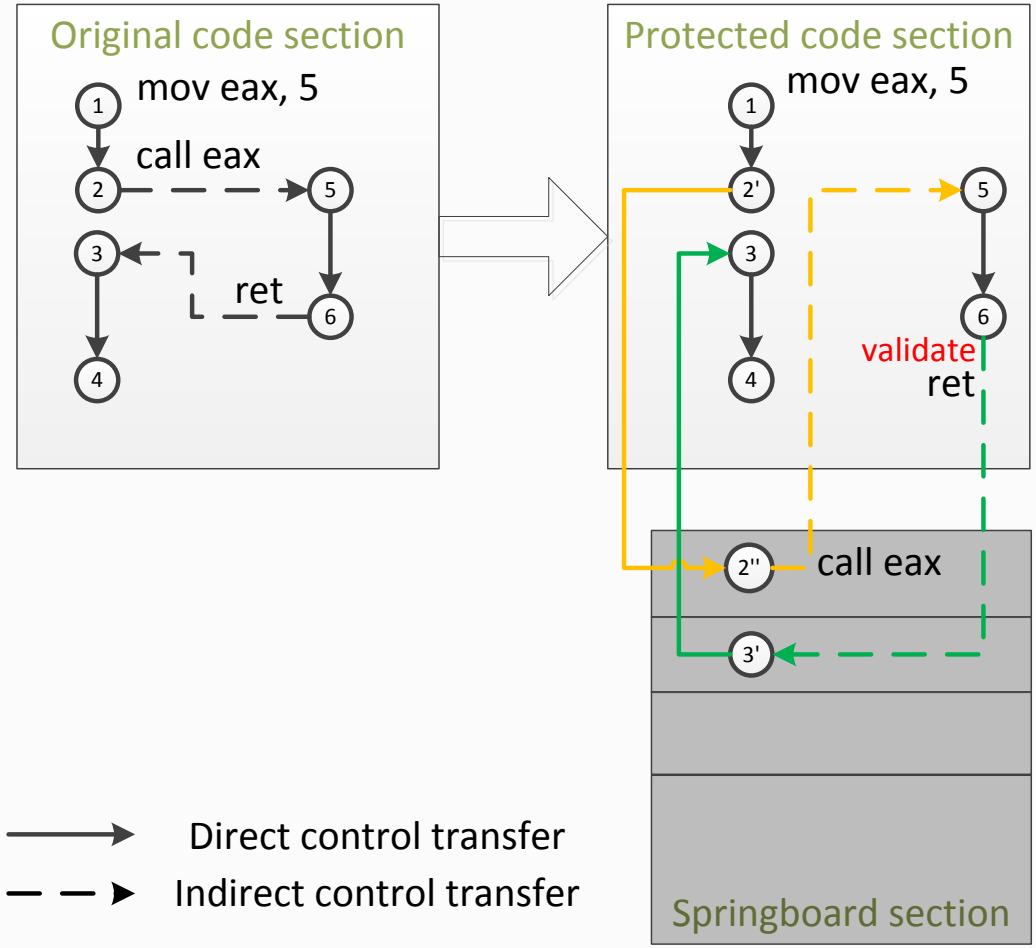
- Policy: all indirect transfers are enforced to aligned stubs in Springboard.



- Validate ret's target.
  - ret-stub in Springboard

BitRewrite: binary rewriting

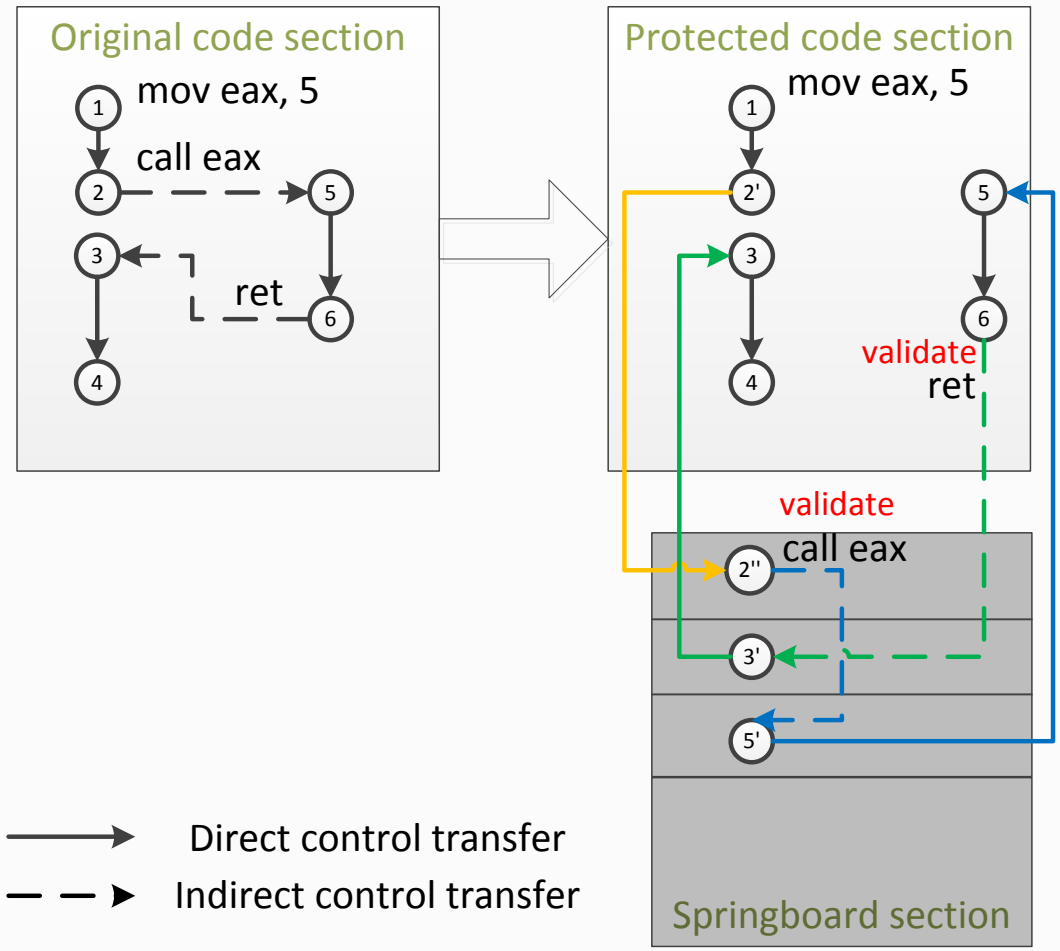
- Policy: all indirect transfers are enforced to aligned stubs in Springboard.



- Validate ret's target.
  - ret-stub in Springboard
- Redirect ret's target.
  - move call to Springboard

BitRewrite: binary rewriting

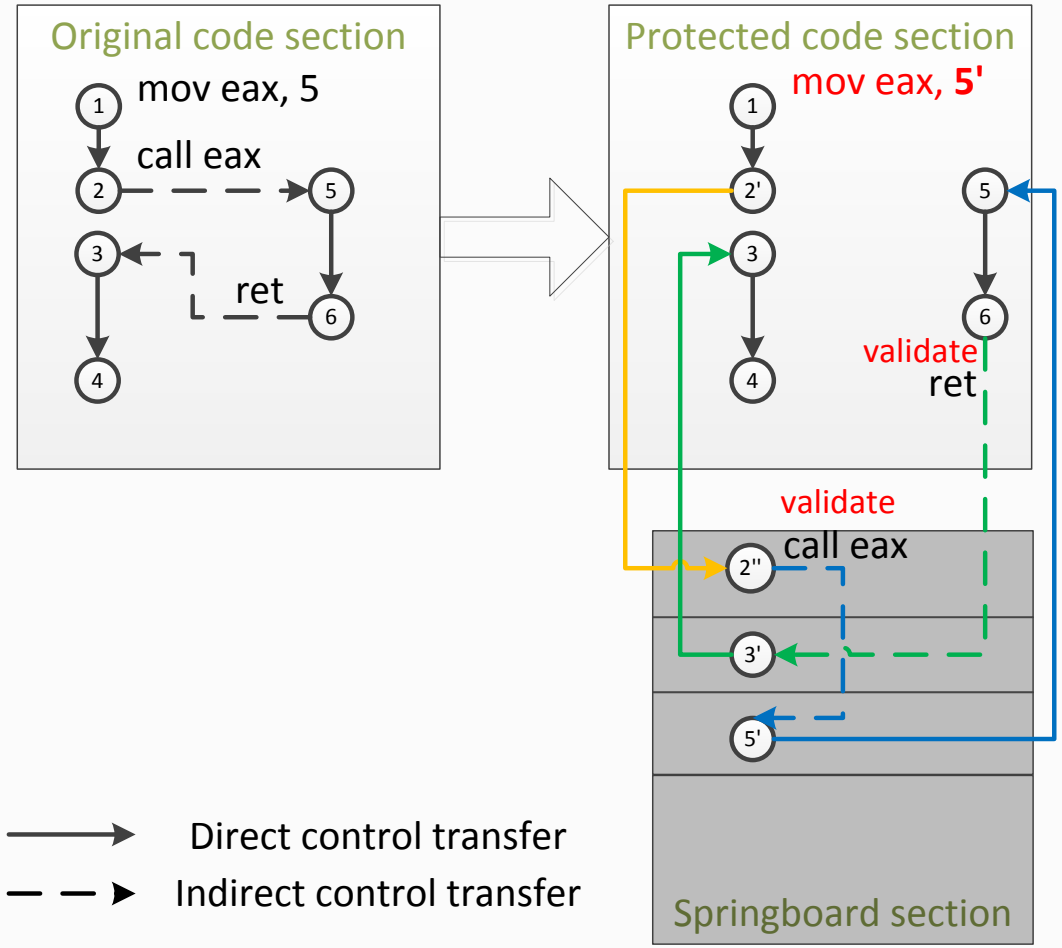
- Policy: all indirect transfers are enforced to aligned stubs in Springboard.



- Validate ret's target.
  - ret-stub in Springboard
- Redirect ret's target.
  - move call to Springboard
- Validate call's target.
  - fp-stub in Springboard

BitRewrite: binary rewriting

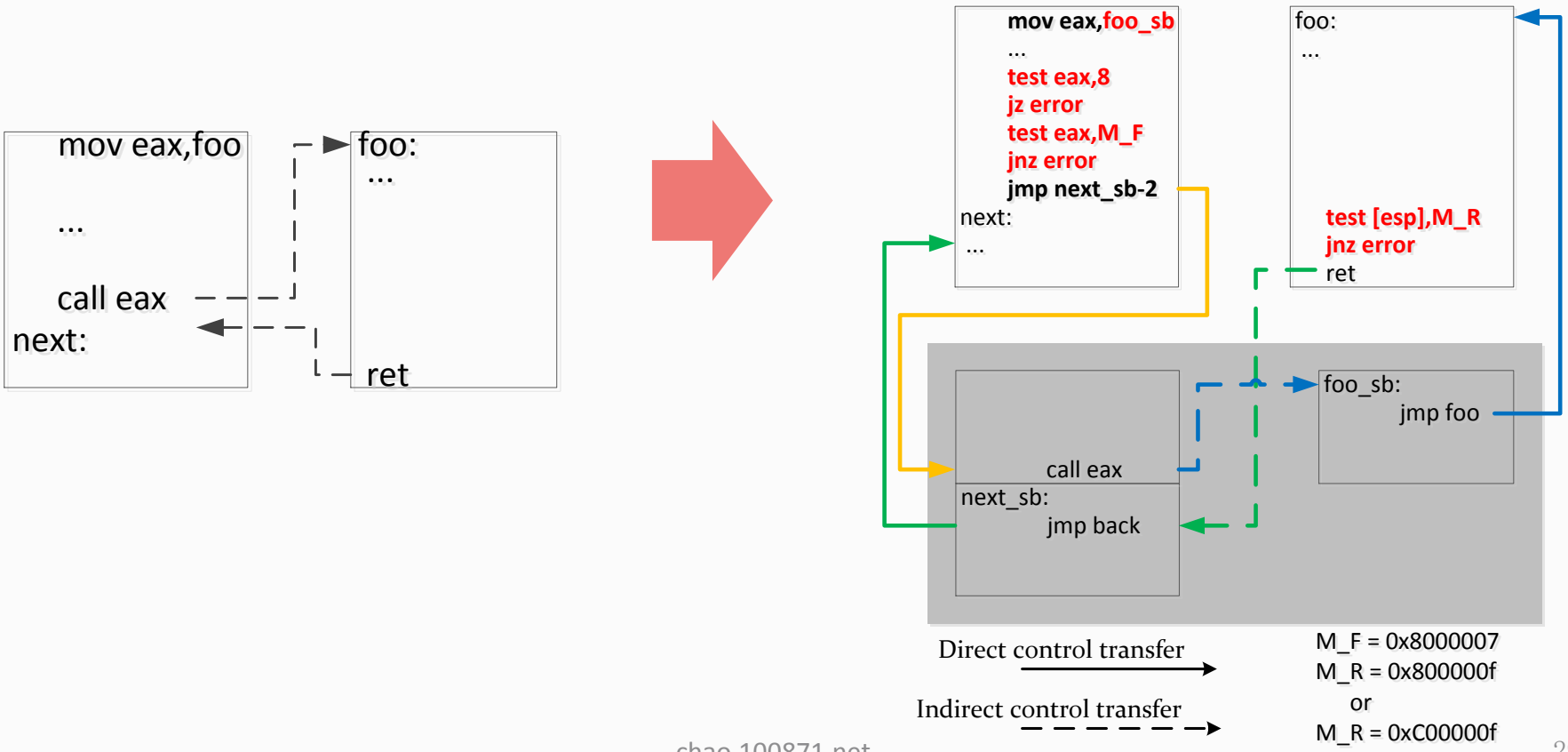
- Policy: all indirect transfers are enforced to aligned stubs in Springboard.



- Validate ret's target.
  - ret-stub in Springboard
- Redirect ret's target.
  - move call to Springboard
- Validate call's target.
  - fp-stub in Springboard
- Redirect call's target.
  - rewrite function pointers

BitRewrite: In a nutshell

- redirect function pointers and return addresses
  - rewrite function pointers in memory
  - move call instructions to springboard
- validate transfer targets with bit testing instructions



BitRewrite: Optimizations

- performance overhead ✓ ✓ ✓
- modularity support ✓ ✓ ✓
- backward compatibility
- source code independent ✓ ✓ ✓

- With the support of W⊗X, we can skip redirecting or validating
  - directly used function pointers
    - call foo
  - directly used imported functions
    - call [imp_foo]
  - switch statements and jump table
    - jmp jump_table[eax*4]

BitRewrite: backward compatibility

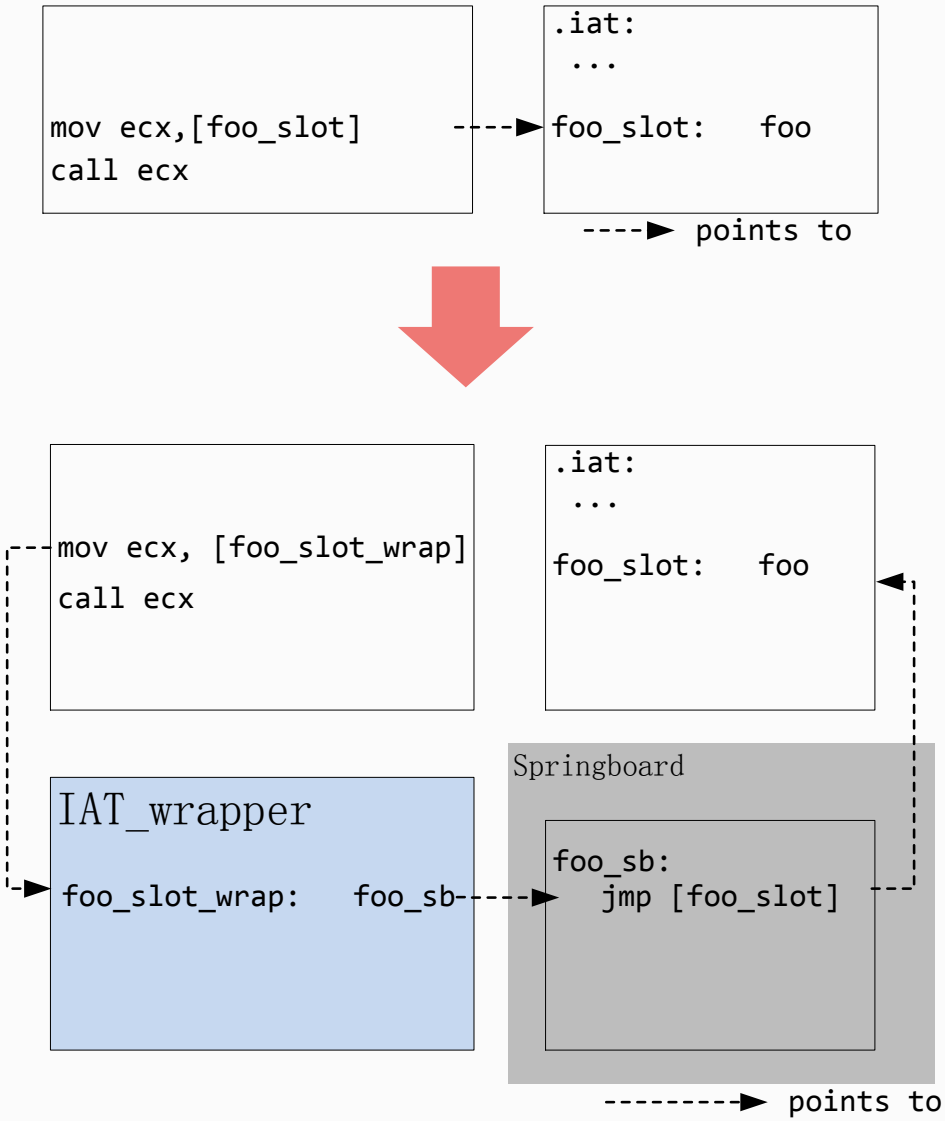
- performance overhead ✓ ✓ ✓
- modularity support ✓ ✓ ✓
- backward compatibility ✓ ✓ ✓
- source code independent ✓ ✓ ✓

- With a full deployment (i.e., all modules hardened),
  - backward compatibility issues no longer existed.

- With incremental deployment, such issues occur when
  - CALL/JMP to functions in unhardened modules
  - RET to return addresses in unhardened modules

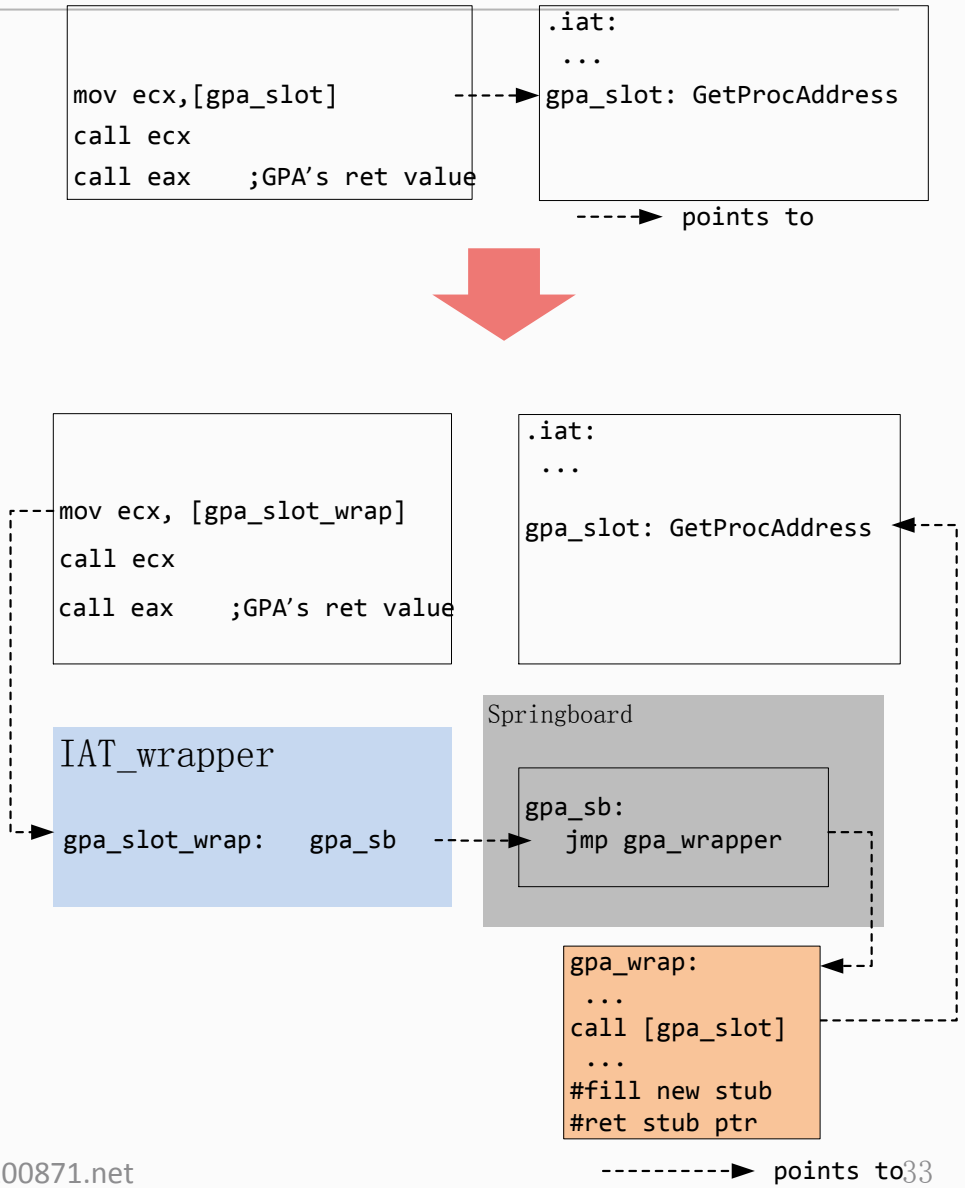
case 1: Imported functions

- Challenge
  - Import Address Table (IAT)
  - load-time resolving
    - the loader will update IAT
    - we cannot statically rewrite IAT entries (function pointers)
- Solution
  - Replace IAT with a wrapper table



case 2: Functions resolved by GetProcAddress

- Challenge
  - GetProcAddress() returns a non-redirected function pointer.
  - The instrumented security check will fail
- Solution
  - wrapper for GetProcAddress
  - redirect return value at runtime



BitVerify: Offline Verification

- whether a given binary conforms to security rules?
  - 27th bit of executable section except Springboard should be 1
  - Code stubs in Springboard are all aligned
  - Function pointers are redirected
  - Call instruction has been redirected
  - Check code has been correctly inserted
  - ...

Outline

- Motivation
- Intuition & Design
- Implementation
- Security Enhancement
- Evaluation
- Conclusion

Policy 1: Restrain sensitive functions

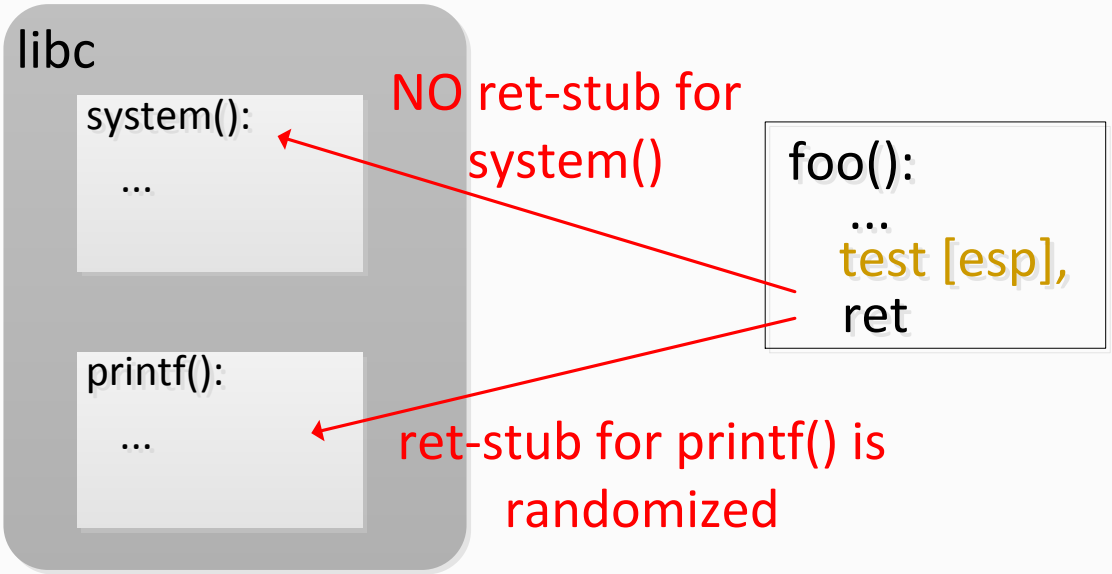
- Sensitive functions
  - system(), execv(), VirtualProtect(), VirtualAlloc(), ...
- Policy:
  - Sensitive functions should be used via direct calls.
    - i.e., their addresses are never taken except hard−coded in instructions
- Benefit:
  - NO fp-stubs for sensitive functions in the Springboard
  - Attackers cannot indirectly call/jump to these functions.

Policy 2: Load-time Randomization

- Policy:
  - The order of all stubs in the Springboard are randomized at load-time.
- Benefit:
  - cannot guess one stub's address even if another address is leaked in the Springboard.
  - A stronger randomization than ASLR.
- It provides an extra layer of protection from the previous CFI-style checking.

Brief Security Analysis

- return-to-libc



(info leakage is needed)

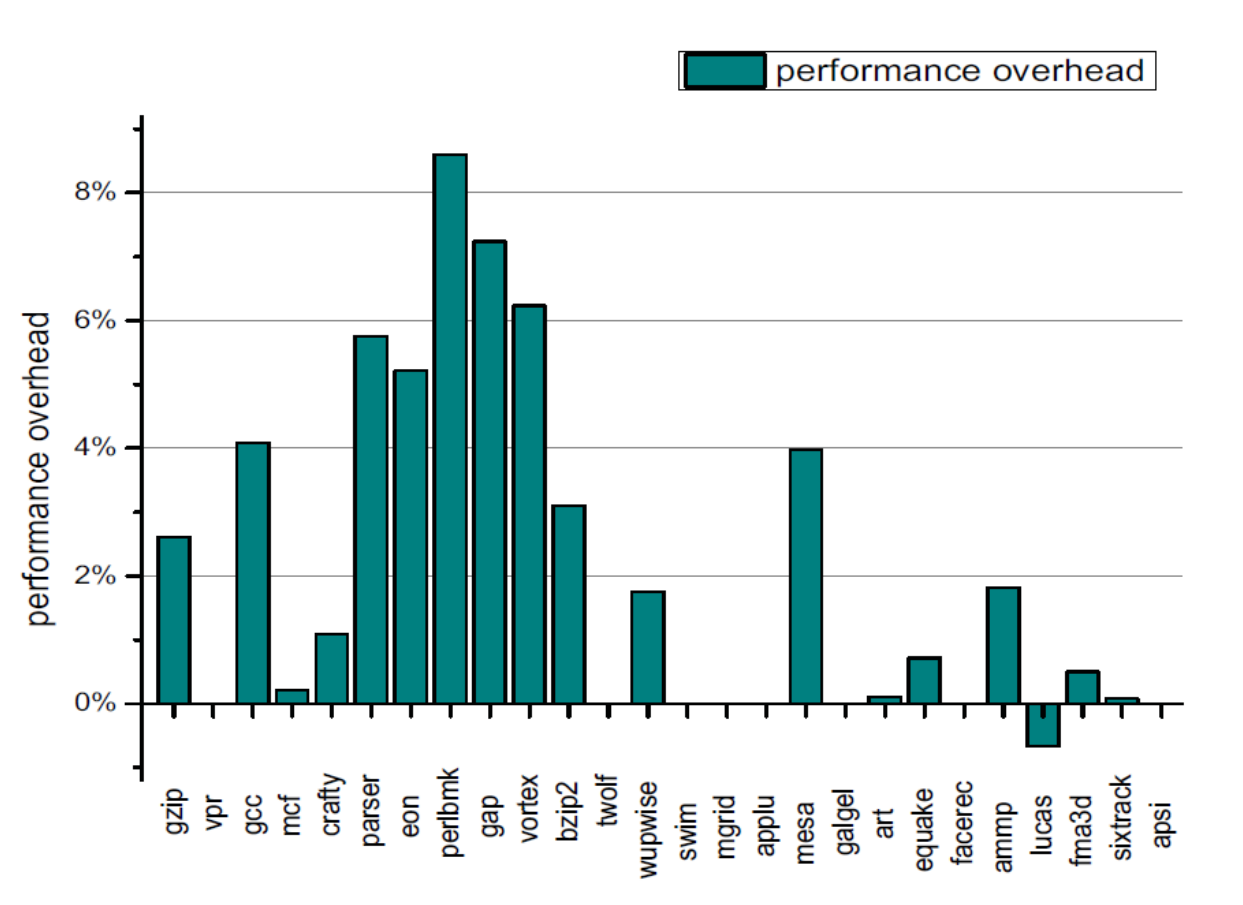
- jump-to-libc
  - similar to return-to-libc
  - gadgets must start from fp-stubs in the Springboard
- ROP
  - Valid gadgets must start from ret-stubs in the Springboard
  - Few valid ROP gadgets
    - much longer, hard to chain.

Outline

- Motivation
- Intuition & Design
- Implementation
- Security Enhancement
- Evaluation
- Conclusion

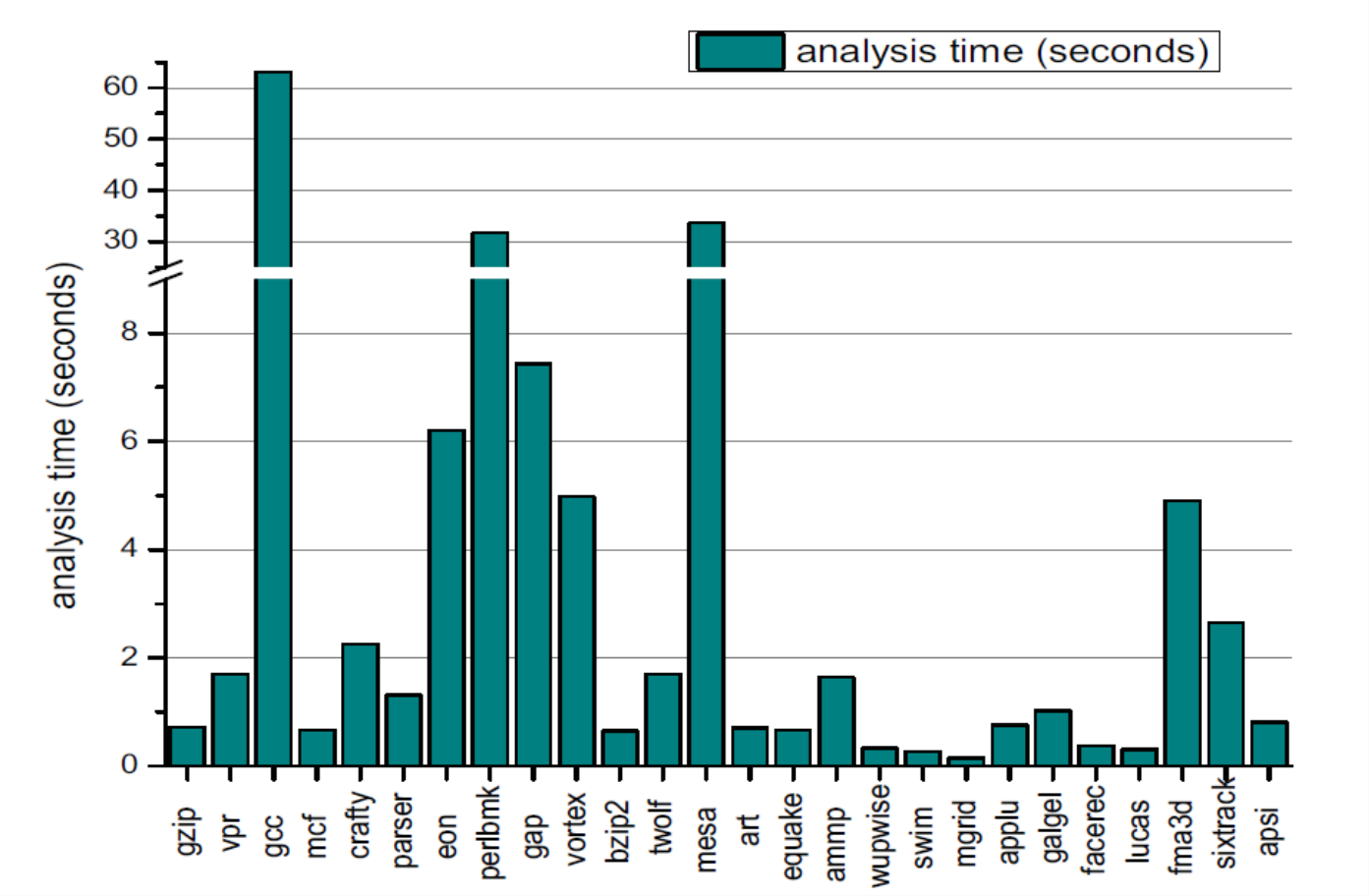
Runtime Overhead

- SPECint2000, average 3.6%, max 8.6%
- SPECfp2000, average 0.59%, max 3.98%



Static Analysis Time

- SPECint2000 & SPECfp2000, 10s seconds



ROP Elimination

- Mona is used to count ROP gadgets
- Valid gadgets must start from ret-stubs in Springboard

SPECfp2000 App	original #gadgets	new #gadgets	valid #gadgets
wupwise	255	0	0
swim	116	134	0
mgrid	161	166	0
applu	182	172	0
mesa	21696	0	0
galgel	1515	952	0
art	1874	0	0
equake	1710	0	0
facerec	826	775	0

ROP gadgets count (part)

Real World Exploits Protection

ID	App	Vul Type	Vul Module	Protected
CVE-2011-0065	**FF 3**	Use After Free	xul.dll	yes
CVE-2010-0249	**IE 6**	Use After Free	mshtml.dll	yes
CVE-2010-3962	**IE 6**	Use After Free	mshtml.dll	yes
CVE-2011-1260	**IE 6**	Mem. Corrupt	mshtml.dll	yes
CVE-2005-1790	**IE 6**	Mem. Corrupt	mshtml.dll	yes
CVE-2008-0348	coolplayer	Stack Overflow	core exe	yes
CVE-2010-5081	RM-MP3	Stack Overflow	core exe	yes
OSVDB-83362	urlhunter	Stack Overflow	core exe	yes
CVE-2007-1195	XM ftp	Format String	core exe	yes
OSVDB-82798	ComSndFTP	Format String	core exe	yes

Outline

- Motivation
- Intuition & Design
- Implementation
- Security Enhancement
- Evaluation
- Conclusion

Conclusion

- Performance and compatibility issues limit the adoption of Control Flow Integrity.
- CCFIR is a lightweight CFI solution providing a strong guarantee, can block control-flow hijacks like ROP and ret2libc.
- CCFIR overcomes the performance and compatibility issues, and make it a practical solution.

Thanks!