Mitigating Code-Reuse Attacks with Control-Flow Locking

Tyler Bletsch, Xuxian Jiang, Vince Freeh

Dec. 9, 2011

NC STATE UNIVERSITY

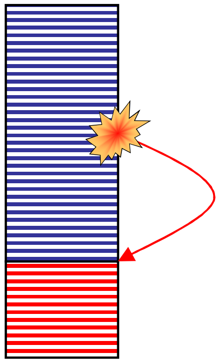
Introduction

- Computer systems run complicated software, which is vulnerable
  - We keep finding new vulnerabilities
  - Vulnerabilities are routinely exploited

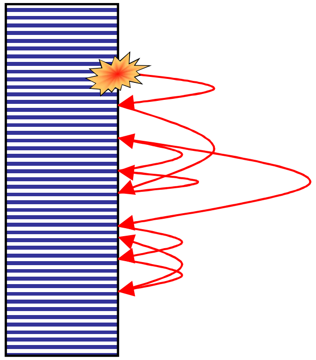
Attack techniques

- Exploit a software vul. to redirect control flow
  - Buffer overflow, format string bug, etc.

  - Code injection attacks
    - Upload malicious machine code
    - Prevented by W^X



  - Code reuse attacks
    - Engage in malicious control flow

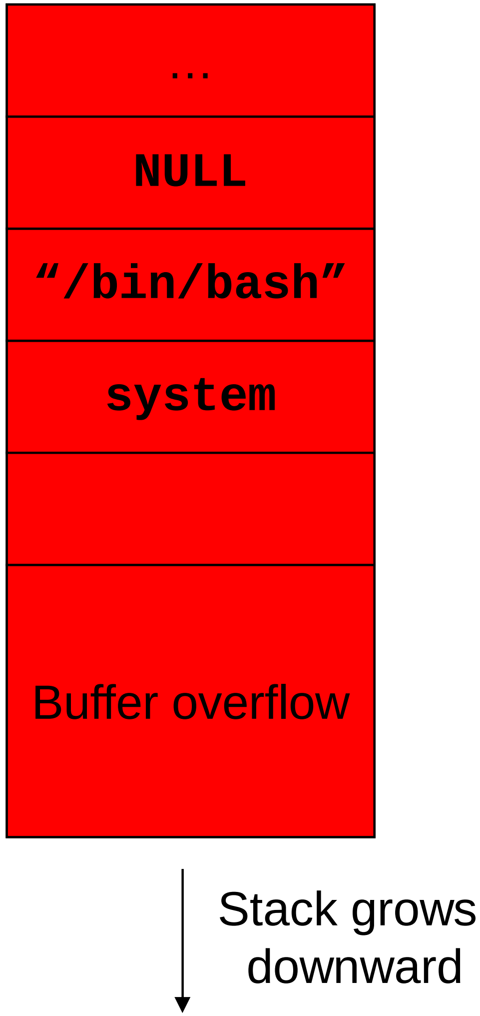


Background on code-reuse attacks

- We assume the attacker can
  - Put a payload into W^X-protected memory
  - Exploit a bug to overwrite some control data (return address, function pointer, etc.)
  - Altered control data will redirect control flow

Background on code-reuse attacks

- Return-into-libc attack
  - Execute entire libc functions
  - Attacker may:
    - Use system/exec to run a shell
    - Use mprotect/mmap to disable W^X
  - Straight-line code only
    - General assumption



Background on code-reuse attacks

- How to get arbitrary computation?
  Return-oriented programming (ROP)

- Chains together *gadgets*: tiny snippets of code ending in `ret`
- Achieves Turing completeness
- Demonstrated on x86, SPARC, ARM, z80, ...
  - Including on a deployed voting machine, which has a non-modifiable ROM
  - Remote exploit on Apple Quicktime[1]

[1] http://threatpost.com/en_us/blogs/new-remote-flaw-apple-quicktime-bypasses-aslr-and-dep-083010

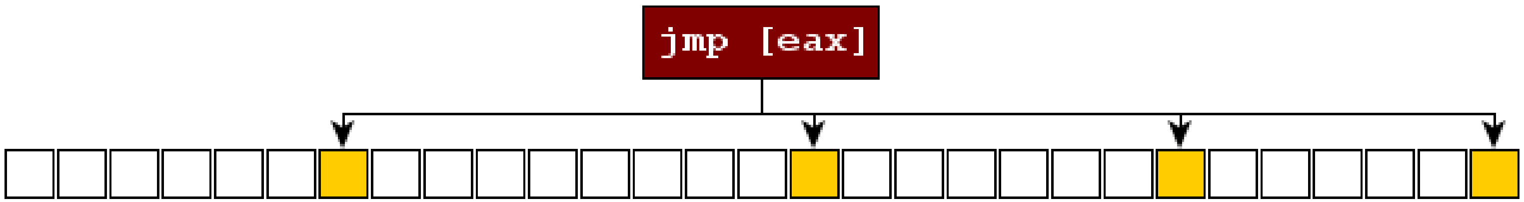
Defenses against ROP

- ROP attacks rely on the stack in a unique way
- Researchers built defenses based on this:
  - ROPdefender[1] and others: maintain a shadow stack
  - DROP[2] and DynIMA[3]: detect high frequency `rets`
  - Returnless[4]: Systematically eliminate all `rets`

- Problem: code-reuse attacks need not be limited to the stack and **`ret`**!
  - Jump-oriented programming[13]: a way to be Turing complete with just **`jmp`**.

Can we do better?

- What is the core problem behind code-reuse attacks?
  - Using control data in memory to allow jumps to literally *anywhere*
- Solution: Constrain attacker choices, move towards finer and finer control flow integrity



Can we do better?

- Earlier work
  - Program shepherding[7]: *instrumentation*-based, up to 7x overhead — Very expensive
  - Control flow integrity[8] (CFI) — Still too expensive
    - Before each transfer, *eagerly* check target for a special token inline with code
    - Relatively high overhead (up to 46%)

- We propose a more efficient mechanism
  - Validation performed *lazily* instead of eagerly
  - Mutex-inspired "locking" mechanism

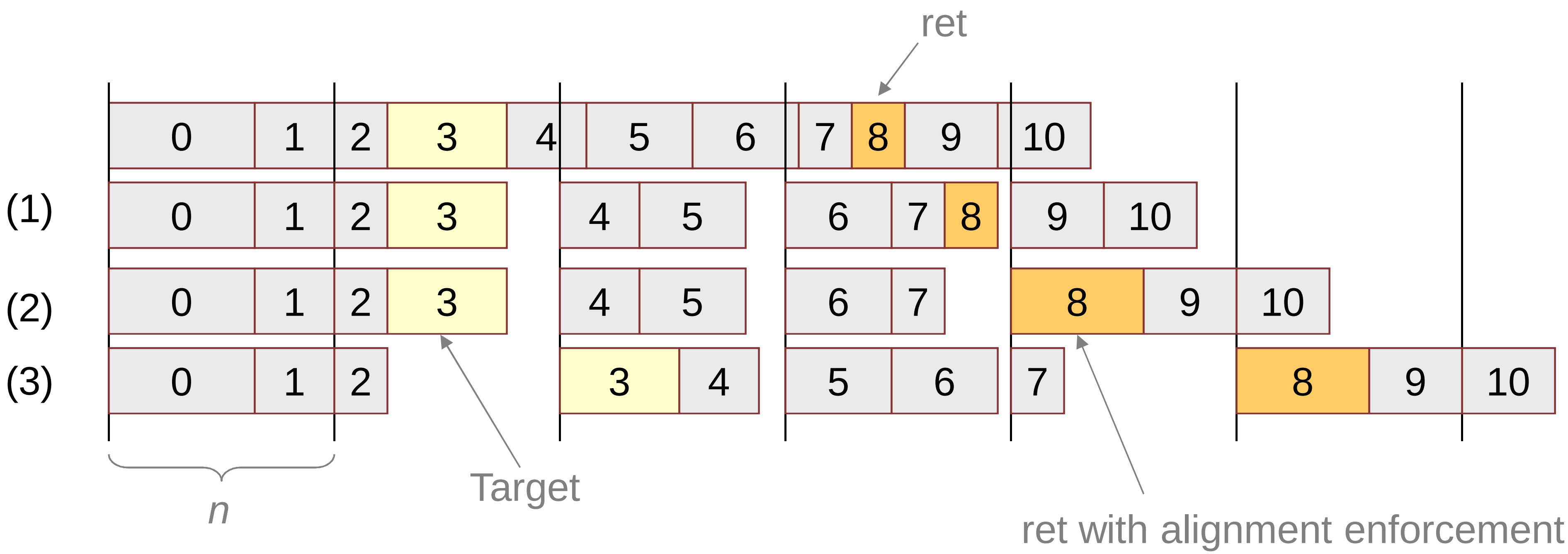
Control flow locking (CFL)

Can we do better?

- *Unintended* code
  - Eliminate it or prevent its execution globally
  - Use a sandboxing technique based on alignment
    - Introduced by McCamant, et al. [10]
    - Developed further in Google Native Client[11]
- *Intended* code

Preventing unintended code

- Impose three changes on compiled code:
  1. No instruction may cross an *n*-byte boundary
  2. All indirect control flow transfers must target an *n*-byte boundary
  3. All targets for indirect control flow transfers must be aligned to an *n*-byte boundary



Can we do better?

- *Unintended* code
  - Prevent its execution globally
  - Use a sandboxing technique based on alignment
    - Introduced by McCamant, et al. [10]
    - Developed further in Google Native Client[11]
- *Intended* code
  - Insert security code at intended control flow transfers
    - Indirect **`jmp`** and **`call`**; all **`ret`** instructions

Handling intended code

- Start with a simple version: Single-bit CFL
  - Before a transfer, insert a "lock":

```
if (k != 0) abort();
k = 1;
```

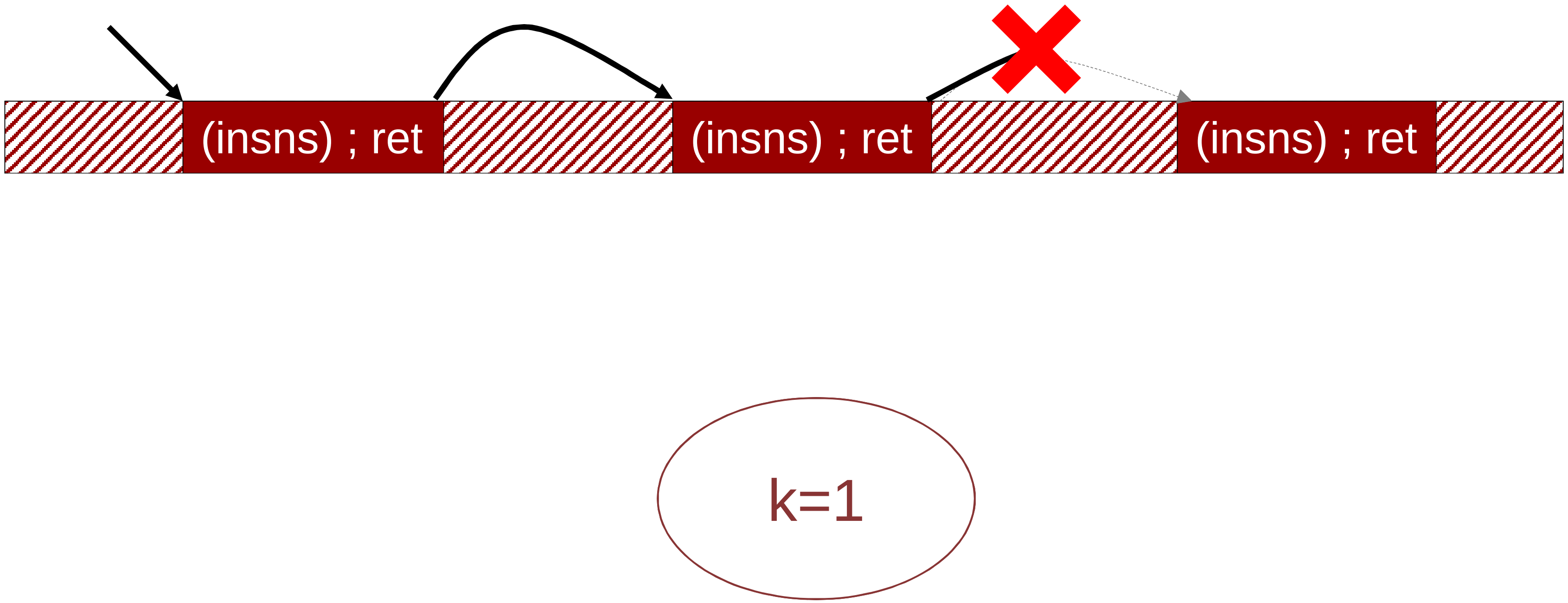
  - Before a "valid target", insert an "unlock":

```
k = 0;
```

Valid target:

- Labels in assembly code that are indirectly callable
- Return sites: locations directly after a call

Effect of single-bit CFL



Improving single-bit CFL

- Control flow forced through valid targets
  - No more gadgets!
  - *Any* valid target unlocks
- We can do better: Multi-bit CFL
  - Assign keys to paths along the control flow graph (CFG)
  - Only the *correct* target unlocks
  - Before a transfer, insert a "lock":

```
if (k != 0) abort();
k = value;
```

  - Before a "valid target", insert an "unlock":

```
if (k != value) abort();
k = 0;
```

Additional considerations

- System calls
  - Insert lock verification code before syscall instructions, e.g.

```
if (k!=0) abort();
```

- Protection of **k**
  - Use x86 segmentation: give **k** its own segment.
  - Ordinary code uses almost no segmentation: there are segment registers never touched by normal code.

Security Analysis

- Cannot violate CFG more than once!
- No syscalls, so what's left?
  - Change some memory
  - Redirect control flow (once)
- But recall our threat model…
  - No new powers!

Threat model

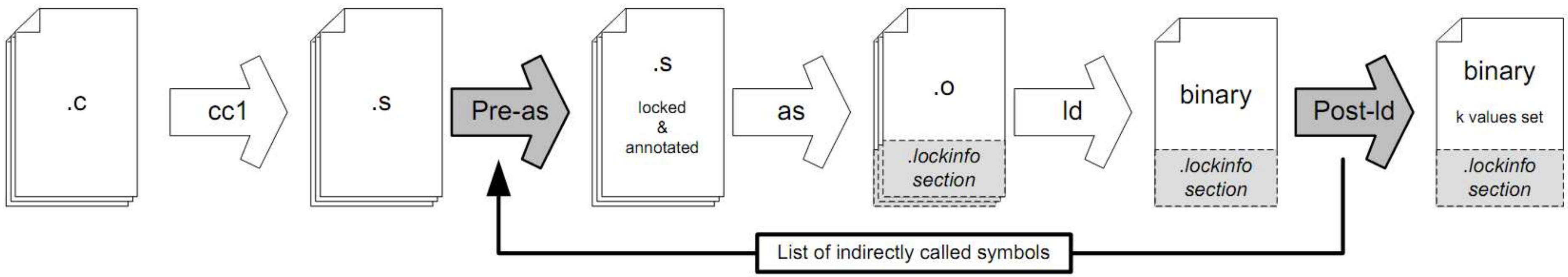
- Attacker can:
  - Overwrite some memory
  - Redirect control flow

Implementation

- Environment:
  - OS: Debian Linux 5.0.4 32-bit x86
  - CPU: Intel Core2Duo E8400 3GHz
  - RAM: 2GB DDR2-800
- Built a CFL-enabled version of:
  - libc (dietlibc)
  - libgcc (helper library included by gcc compiler)
  - Application under test
- Based on statically linked binaries

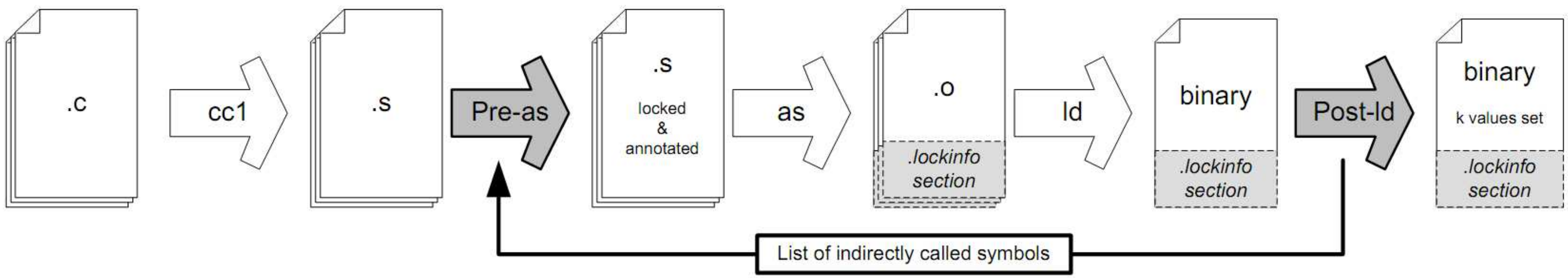
Implementation

- Added two phases to normal gcc build system:
  - Pre-assembly phase: Rewrites assembly code
  - Post-link phase: Extracts CFG, patches up binary



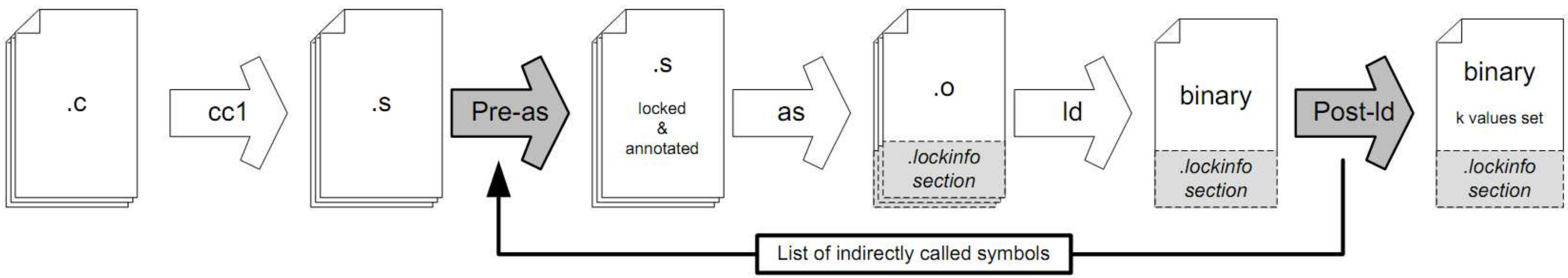
Pre-assembly phase

- The pre-assembly rewriter will:
  1. Do unintended code prevention, n=32 bytes
  2. Insert lock code before all indirect control transfers
  3. Insert unlock code at all indirect control targets
  4. In a section called "**`.lockinfo`**", make note of:
     - All symbols and code label references
     - All direct calls and indirect control flow transfers
     - Location of all lock & unlock code
- Lock/unlock code has dummy values for **k**.



Post-link phase

- The post-link phase will:
  1. Use the **`.lockinfo`** to identify:
     - All lock and unlock code locations
     - All referenced code symbols (i.e., indirectly callable symbols)
     - The CFG
  2. Export the list of indirectly callable symbols
  3. Compute & patch the **k** values of lock and unlock code directly into the finished binary



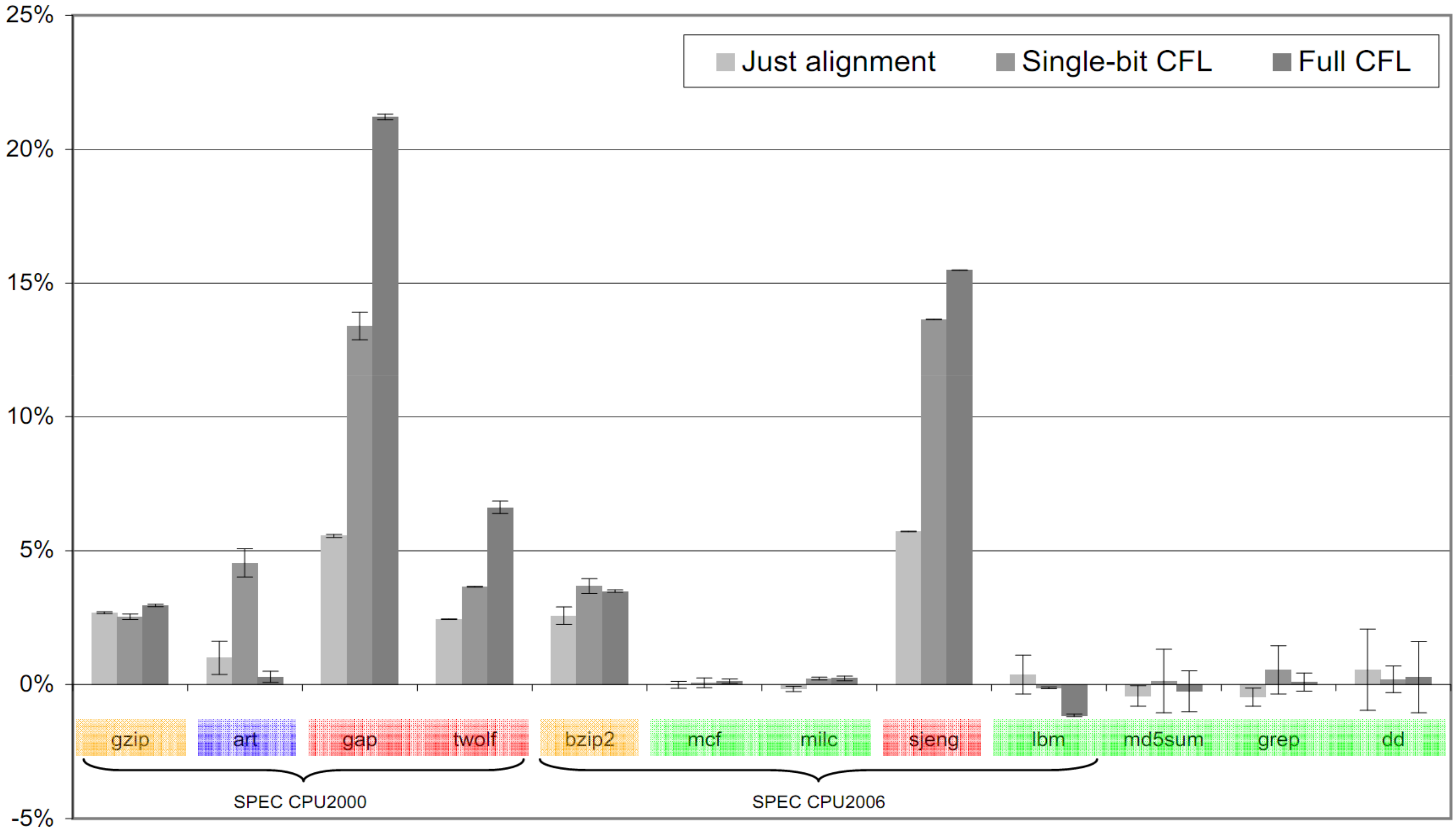
Evaluation

- Correctness
  - "Reliable disassembly"
    - Introduced in Google Native Client project
    - A natural consequence of alignment technique
    - Because unintended code is removed, we can reliably walk the disassembly
  - Verify that all control flow transfers are preceded by lock code
- Performance

Performance evaluation setup

- Workloads:
  - Several from SPEC CPU 2000 and 2006
  - Selected UNIX utilities
- Levels of protection:
  - None: No changes made
  - Just alignment: Add only the alignment shims to preclude unintended code
  - Single-bit CFL: Implement the simple CFL scheme we introduced first
  - Full CFL: The complete CFL scheme
- Overhead: slowdown of the latter three versus "None".

CFL overhead in various workloads



Discussion

- CFL will constrain execution to the CFG, allowing one violation at most

- It is only as good as the CFG it enforces

- “Non-control-data attacks are realistic threats”[12]

Conclusion

- Control flow locking
  - Defends against code-reuse attacks
  - Checks *lazily* rather than *eagerly*
  - Low overhead, competitive performance

Questions?

References

[1] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. Technical Report HGI-TR-2010-001, Horst Gortz Institute for IT Security, March 2010.

[2] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In 5th ACM ICISS, 2009

[3] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic Integrity Measurement and Attestation: Towards Defense against Return-oriented Programming Attacks. In 4th ACM STC, 2009.

[4] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with return-less kernels. In 5th ACM SIGOPS EuroSys Conference, Apr. 2010.

[5] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In 14th ACM CCS, 2007.

[6] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-Oriented Programming Without Returns. In 17th ACM CCS, October 2010.

[7] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution Via Program Shepherding. In 11th USENIX Security Symposium, August 2002.

[8] Martin Abadi, Mihai Budiu, Ulfar Erilingsson, and Jay Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In 12th ACM CCS, October 2005

[9] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In ACSAC, 2010.

[10] Stephen McCamant and Greg Morrisett. Efficient, verifiable binary sandboxing for a CISC architecture. In MIT Technical Report MIT-CSAIL-TR-2005-030, 2005.

[11] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. Communications of the ACM, 53(1):91–99, 2010.

[12] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In In USENIX Security Symposium, pages 177–192, 2005.

[13] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, Zhenkai Liang, "Jump-Oriented Programming: A New Class of Code-Reuse Attack," Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2011), Hong Kong, China, March 2011.