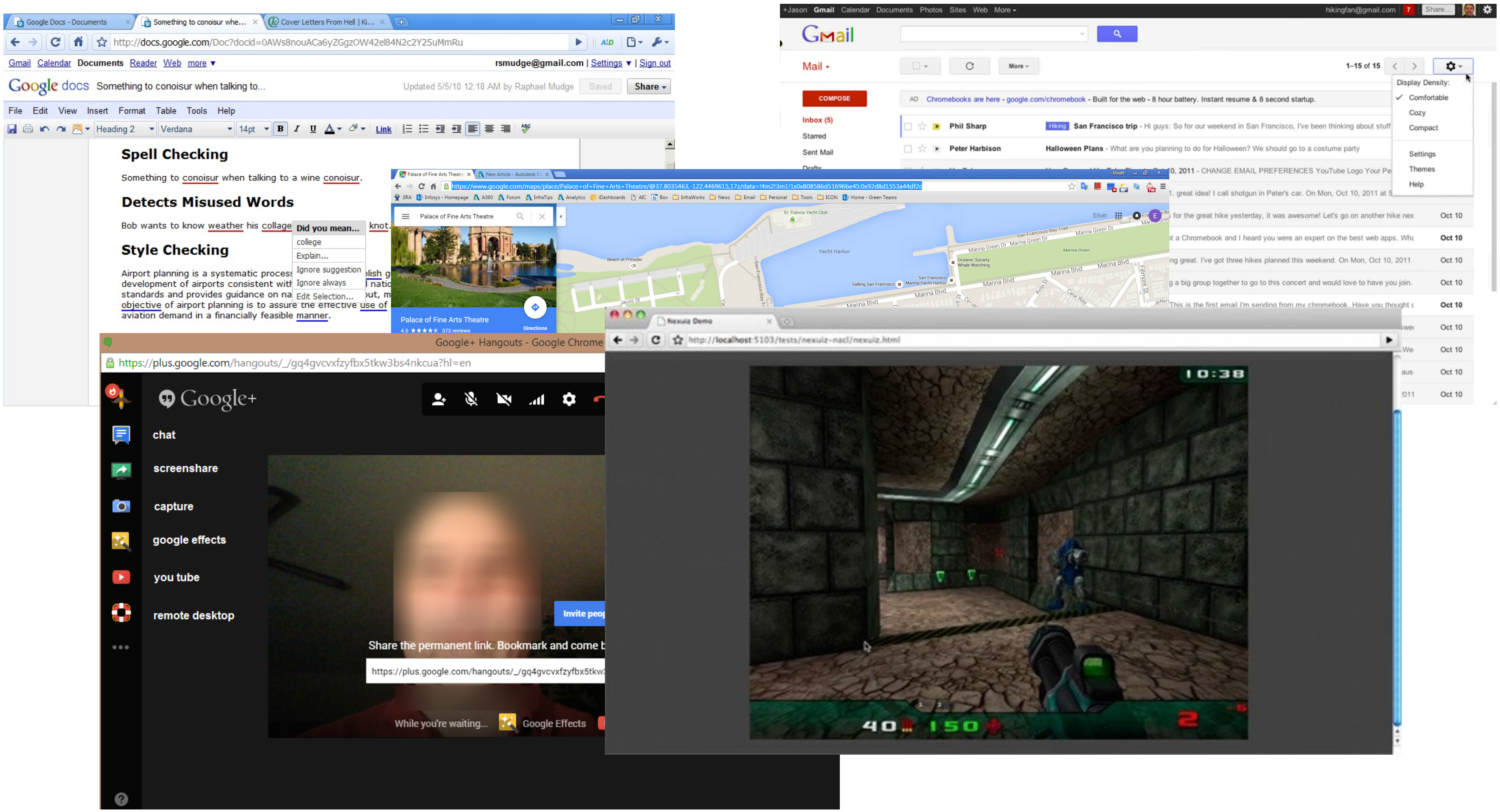
Automatic Software Security Hardening

Gang Tan

School of EECS, Penn State Univ.

INSR Industry Day, *Apr 24th, 2017*

Web Browsers:
Rich Application Platforms

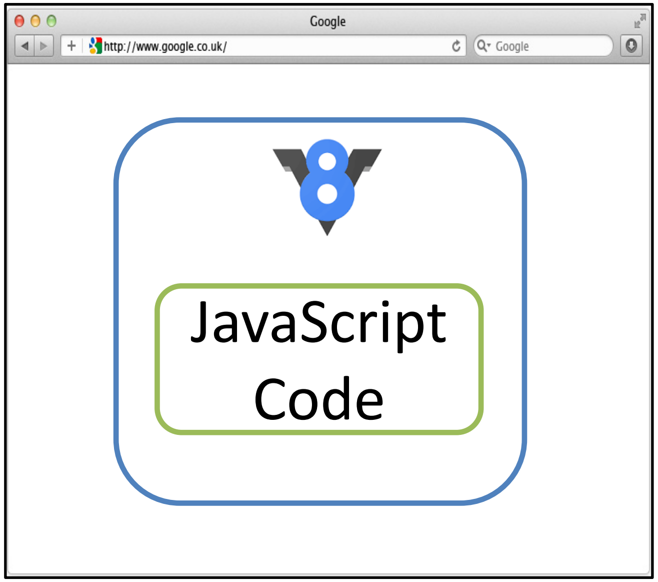


Browser Extensions (Plug-ins)

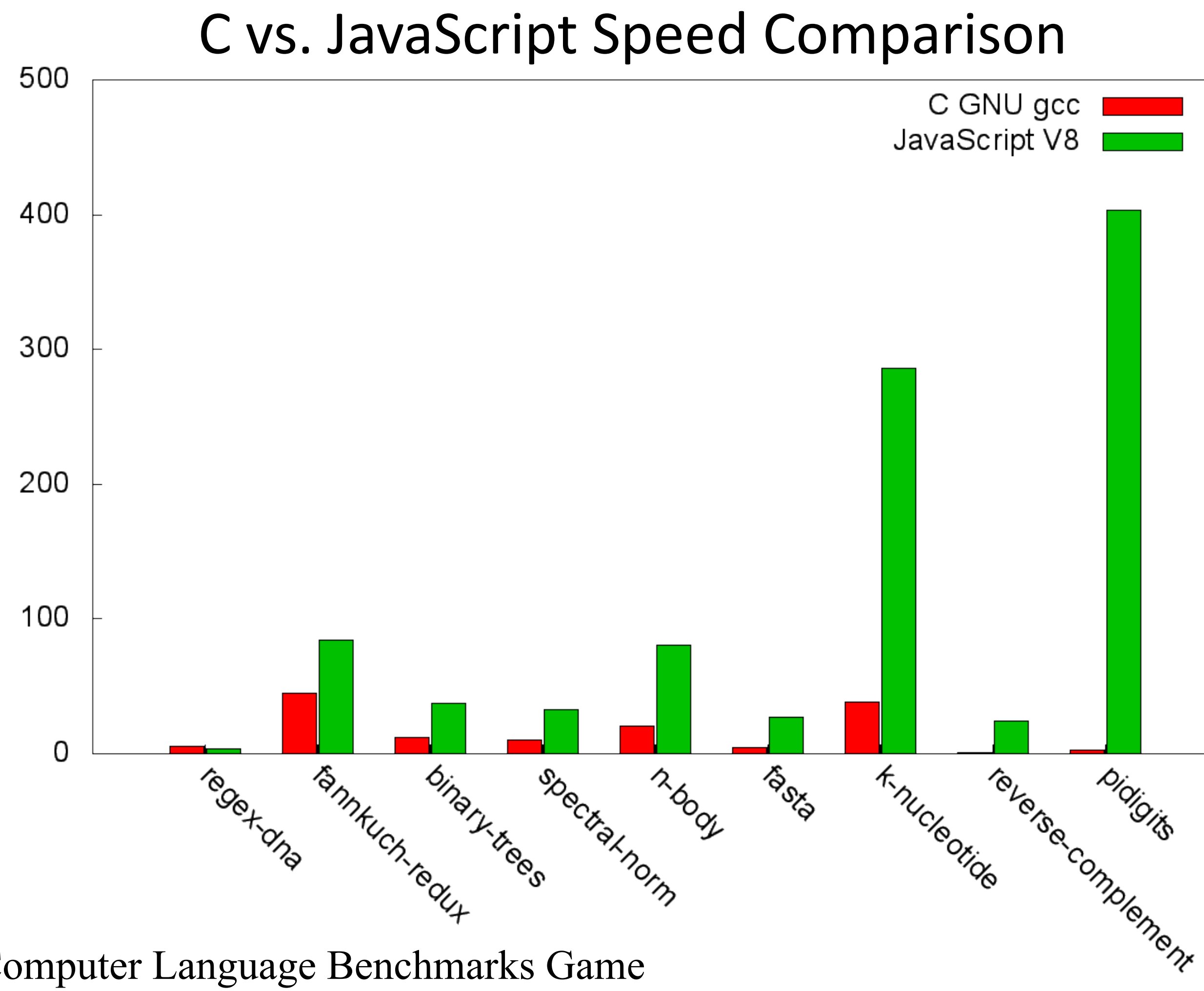
- Extend the functionality of a browser
  - E.g., email client, pdf viewer, …
- All major browsers allow extensions
  - Developed by third-party vendors
  - Communicate with the browser kernel via an interface (NPAPI/PPAPI)
- Security and privacy concerns?
  - Extensions in the same address space as the browser
  - Malicious/buggy extensions can crash the browser, corrupt the browser state, or leak sensitive information

One Solution: Write Extensions in a Safe Language (JavaScript)

- The JavaScript execution engine restricts the behavior of JavaScript code
  - Interpret and monitor JavaScript code for security and privacy violations
    - No direct access to the internal browser state
    - Privileged operations are checked
  - E.g., Chrome's V8 JavaScript engine



However, Performance Concern



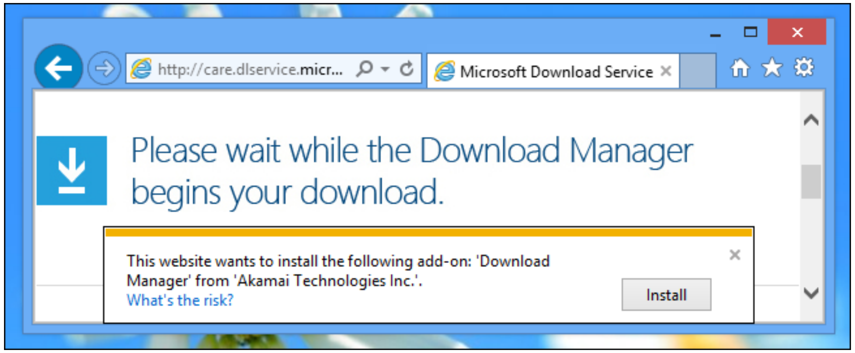
Source: The Computer Language Benchmarks Game

What is Desired in Writing Browser Extensions?

- Develop extensions in **any language**
  - Including C/C++
- Important
  - When performance is critical
    - E.g., graphics-intensive video games
  - When incorporating legacy code developed in other languages
    - No need to rewrite it in JavaScript

Internet Explorer's ActiveX Controls

- Allow IE to install native-code extensions
- No security provided
  - Native extensions run without any constraint
- Ask users before installation
  - Delegate security to users never a good idea



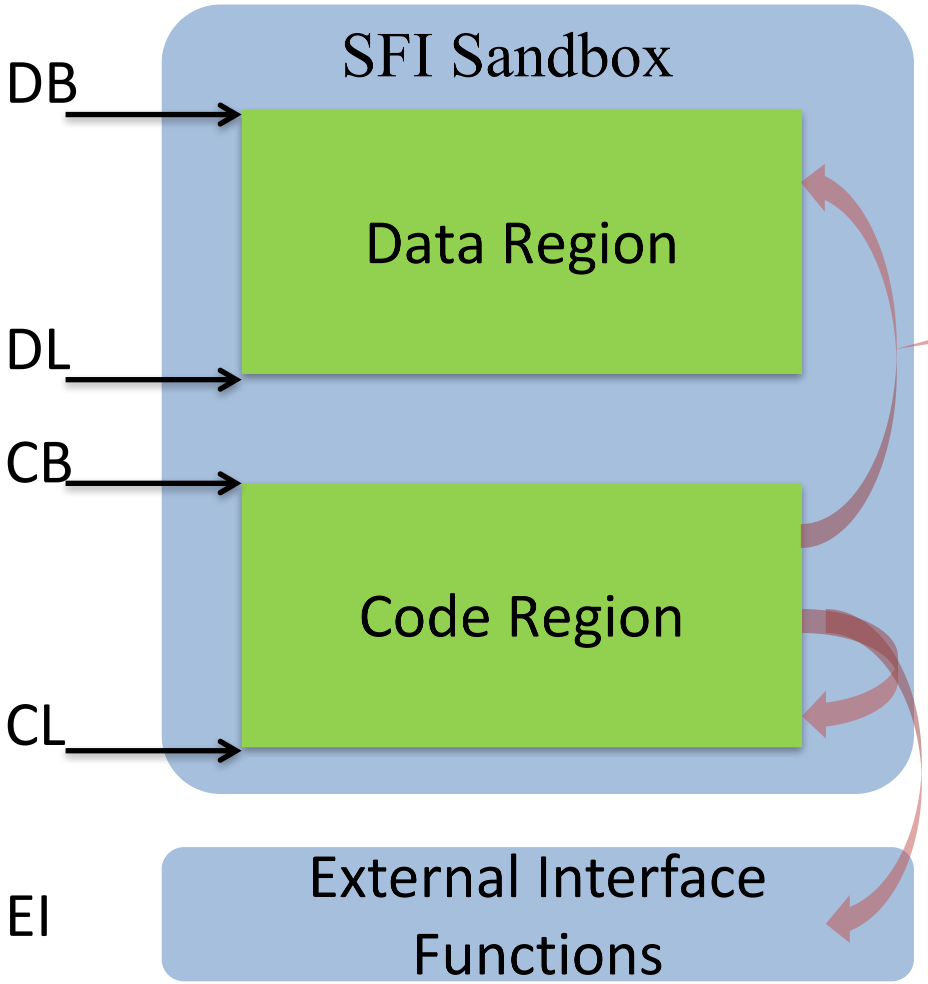
Chrome's Native Client

- Safely running native-code extensions in Chrome
  - Security: a **sandbox** around an extension
  - **Much better performance** than JavaScript
  - Accommodate **legacy code**

NaCl's Sandboxing Mechanism

- Based on Software-based Fault Isolation (SFI)
  - [Wahbe *et al.* SOSP 1991]
- Establish a logical sandbox around an extension
  - The sandbox is in a pre-specified memory-address range
  - Sandbox enforced through **automatic rewriting** of extension code
    - Insert checks before dangerous operations

The SFI Policy



Data aspect:
All mem reads/writes remain in [DB, DL]

Control-flow aspect:
All jumps targets in [CB,CL] or EI

Enforcing the SFI Policy

- Use a compiler to insert checks into the program before dangerous instructions (reads, writes, and jumps)

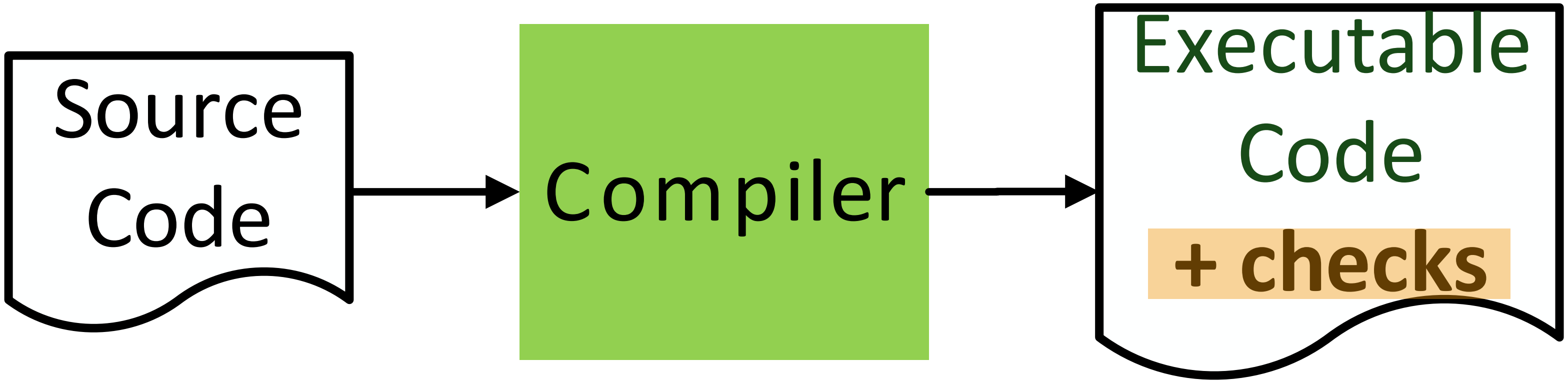
```
mem(a+12) := b        //unsafe write
```

```
if (a+12) < DB then goto exit
if (a+12) > DL then goto exit
mem(a+12) := b
```

```
// more efficient
c := mask(a+12)
mem(c) := b
```

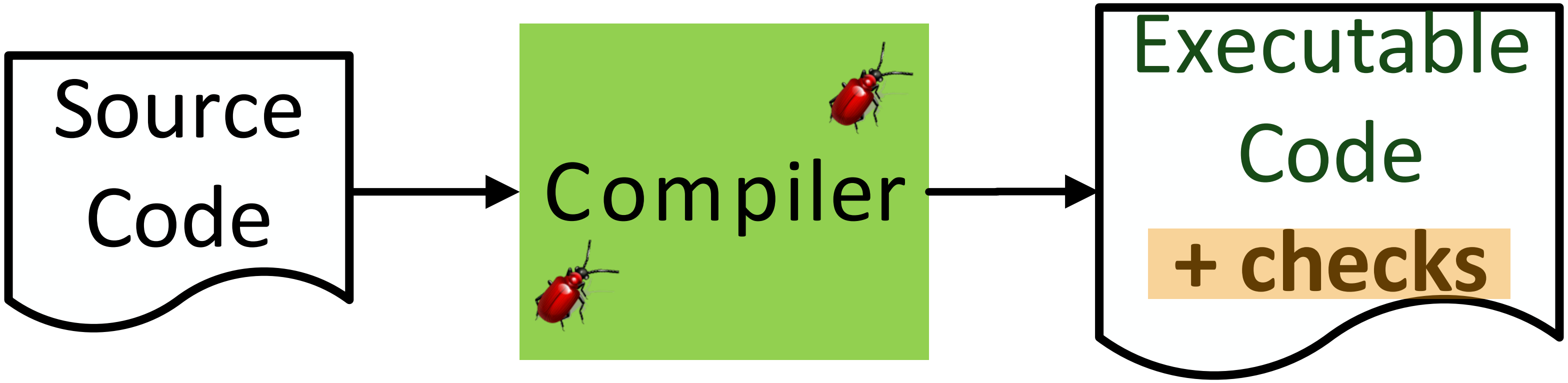
The masking forces c to be in the data region

Automatic Software Hardening



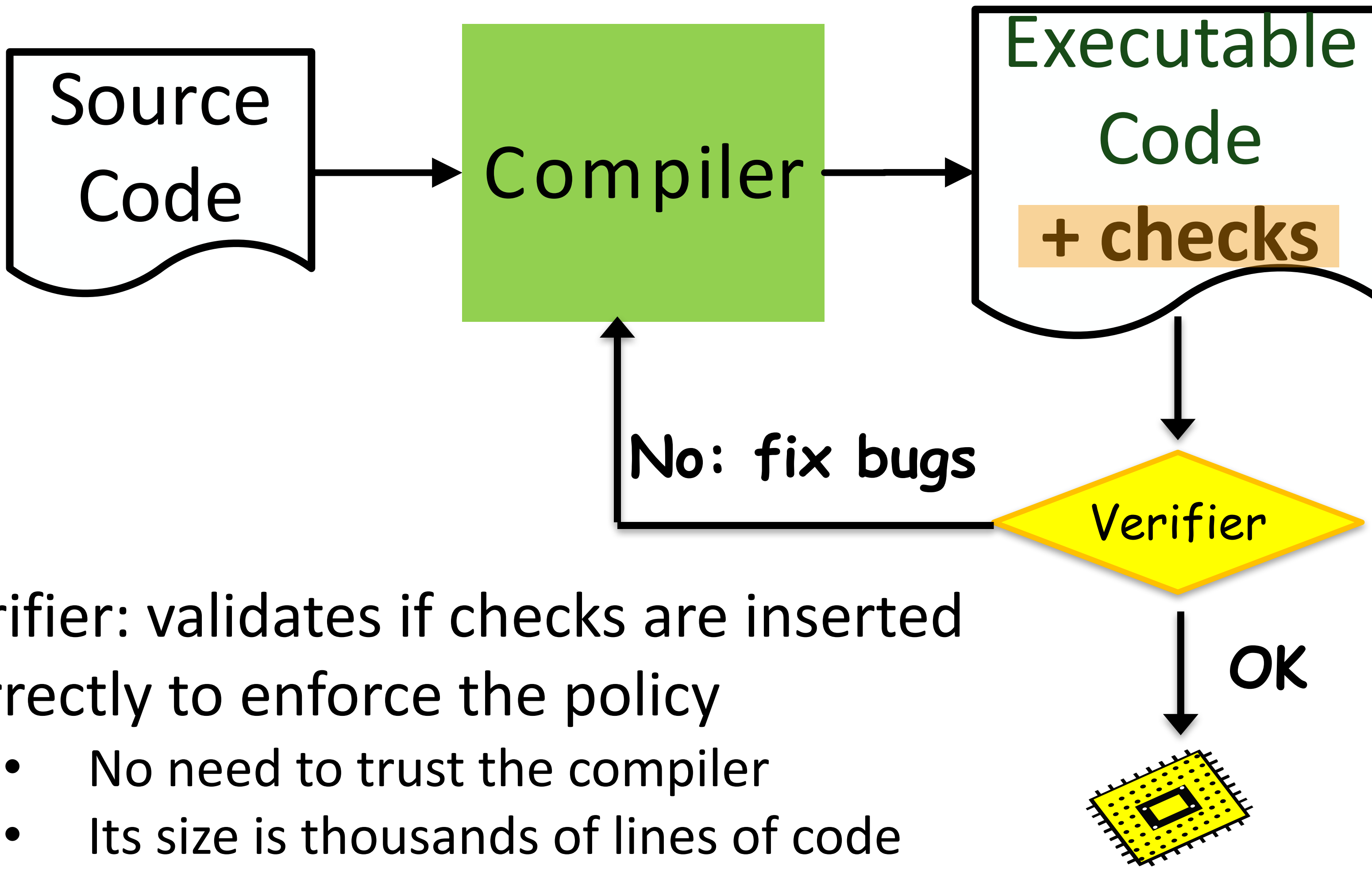
- Perform **program transformation** to embed security checks into the executable code
  - Detect attacks during runtime
- Low performance overhead
  - No context switch (reference monitor is inlined)
  - Security checks can be optimized using static analysis
    - Remove/move checks [Zeng, Tan, Morrisett CCS 2011]
- Can enforce any safety policy such as SFI [Schneider 1998]

Can We Trust the Compiler?



- Compilers may be buggy
  - It may insert/optimize checks in a wrong way
- NaCl uses a modified gcc compiler
  - 7.3 million lines of code, as of 2012
- Hundreds of compiler bugs found in recent work
  - [Yang *et al.* PLDI 2011], [Wang *et al.* SOSP 2013]

Trust, But Verify



Verifier: validates if checks are inserted correctly to enforce the policy

- No need to trust the compiler
- Its size is thousands of lines of code

Now, Can We Trust the Verifier?

- As security researchers, we need to be paranoid …
- Google NaCl's verifier
  - It checks if an input binary satisfies the SFI policy
  - Pile of C code with a manually written decoder for binaries
- A bug in the verifier could result in a security breach
  - Google ran a security contest early on NaCl: bugs found in its verifier!

Question: How to construct high-fidelity verifiers?

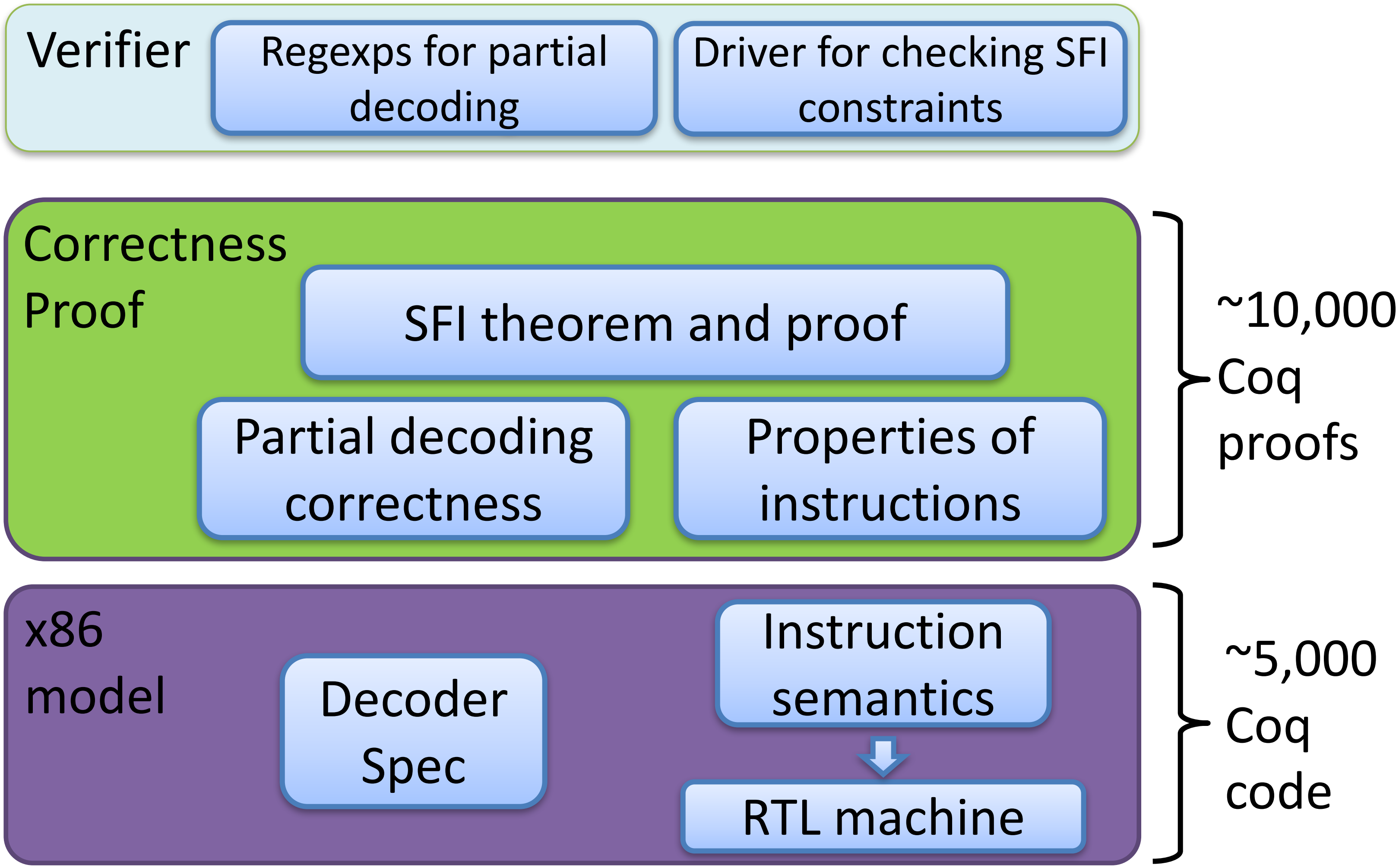
Verifying the Verifier

- Goal: **a provably correct verifier**
- Theorem: if some binary passes the verifier, then the execution of the binary should obey the intended SFI policy

RockSalt [Morrisett, Tan, Tassarotti, Gan, Tristan PLDI 2012]

- A new SFI verifier for x86-32
- **Smaller**
  - Google: manually written code for partial decoding ; plus 600 lines of C driver code
  - RockSalt: regexps for partial decoding ; plus 80 lines of C driver code
- **Faster**: on 200Kloc of C
  - Google's: 0.9s
  - RockSalt: 0.2s
- **Stronger**: RockSalt is proven correct
  - The proof is machine checked in an interactive theorem prover (Coq)

RockSalt Architecture

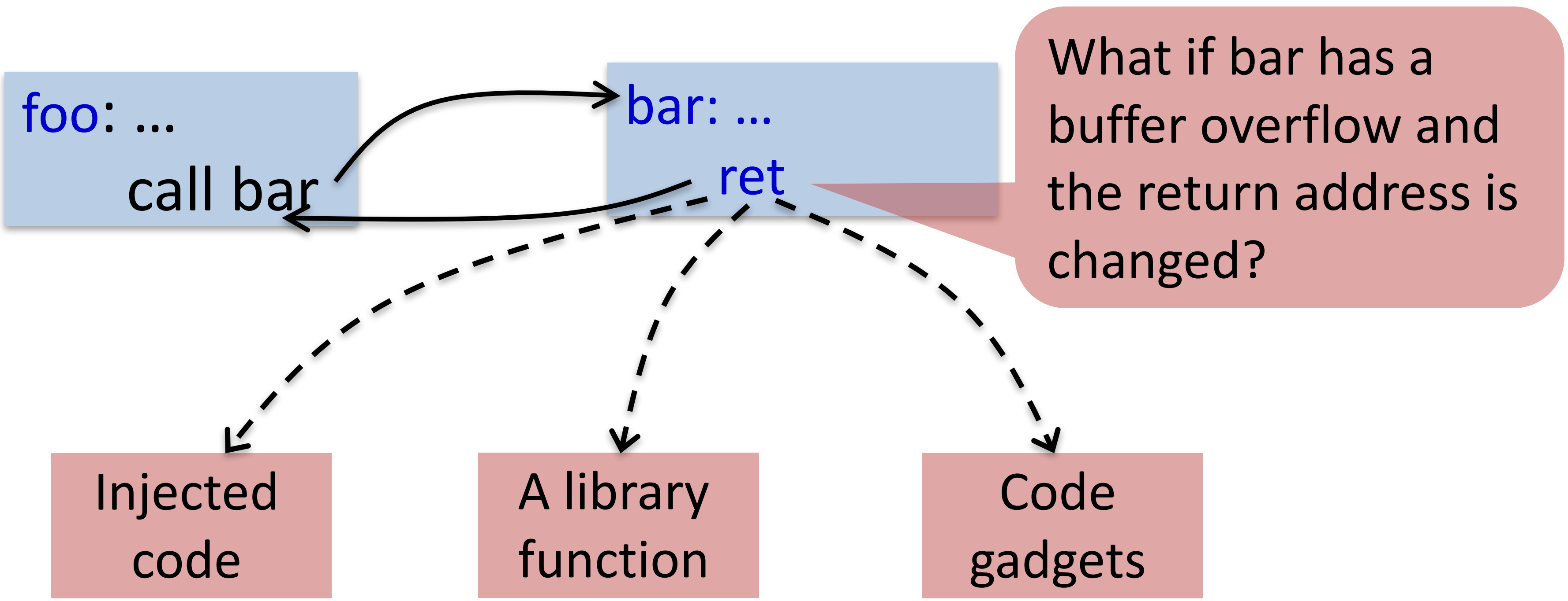


Going Beyond Fault Isolation

- More advanced properties can be enforced via software hardening
  - **Control-Flow Integrity (CFI)**
  - Data-Flow Integrity (DFI)
  - Fine-grained memory-access control
  - Memory safety
  - Taint tracking
  - …

Control-Flow Integrity: Preventing Control-Flow Hijacking Attacks

Example of Control-Flow Hijacking



Stack smashing

Return to libc

Return-Oriented Programming (ROP) attacks

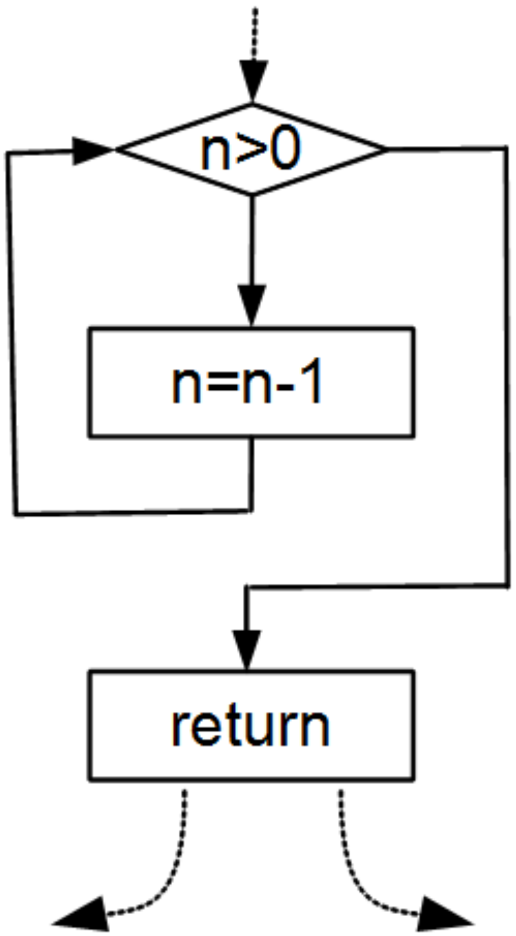
Control Flow Integrity (CFI) [Abadi *et al.* CCS 2005]

1) Pre-determine a control-flow graph (CFG) of a program
2) Enforce the CFG by instrumenting **indirect branches** in the program
   - Instrumentation: insert checks before indirect branches
   - Indirect branches include returns, indirect calls, and indirect jumps

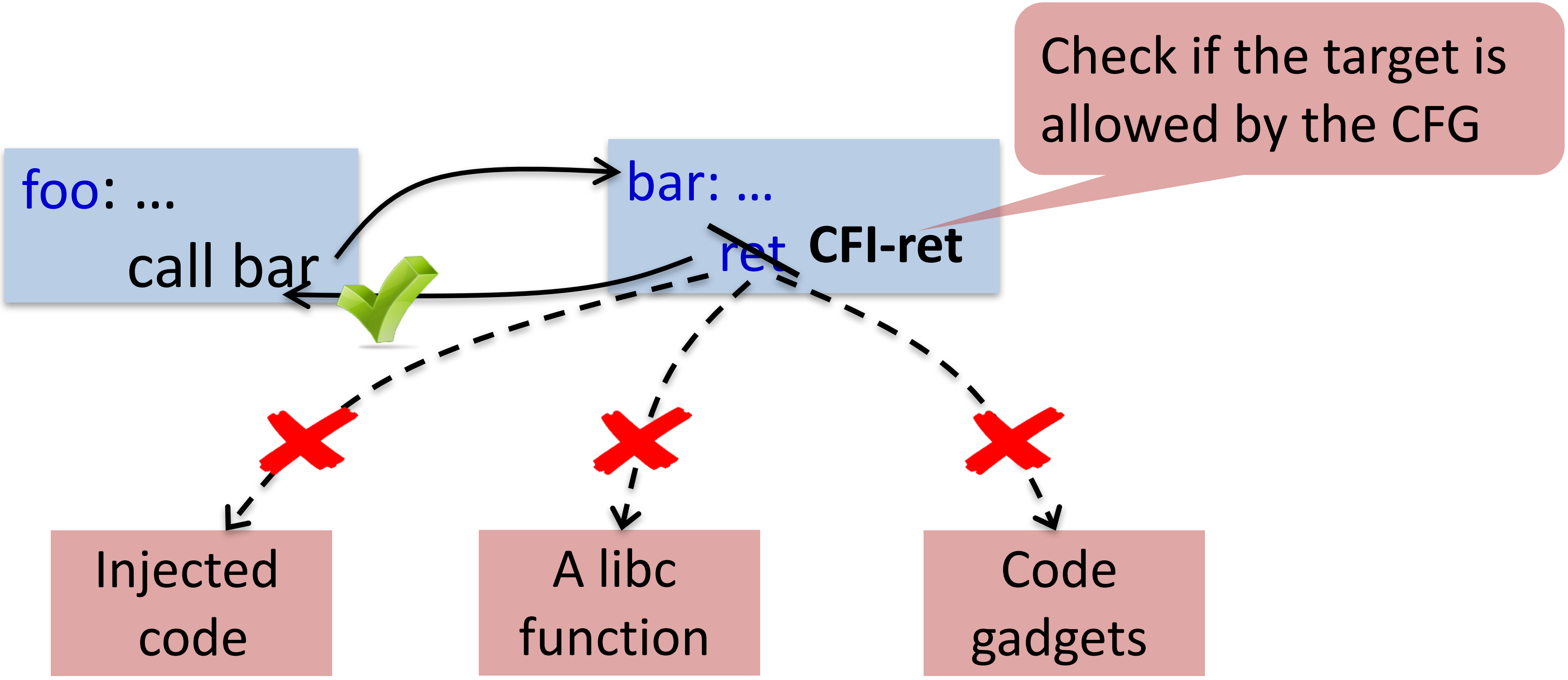
CFI Policy: execution of the instrumented program follows the pre-determined CFG, even under attacks

Control Flow Graphs (CFG)

- **Nodes** are addresses of basic blocks of instructions
- **Edges** connect control instructions (jumps and branches) to allowed destination basic blocks



CFI: Mitigating Control-Flow Hijacking



Stack smashing

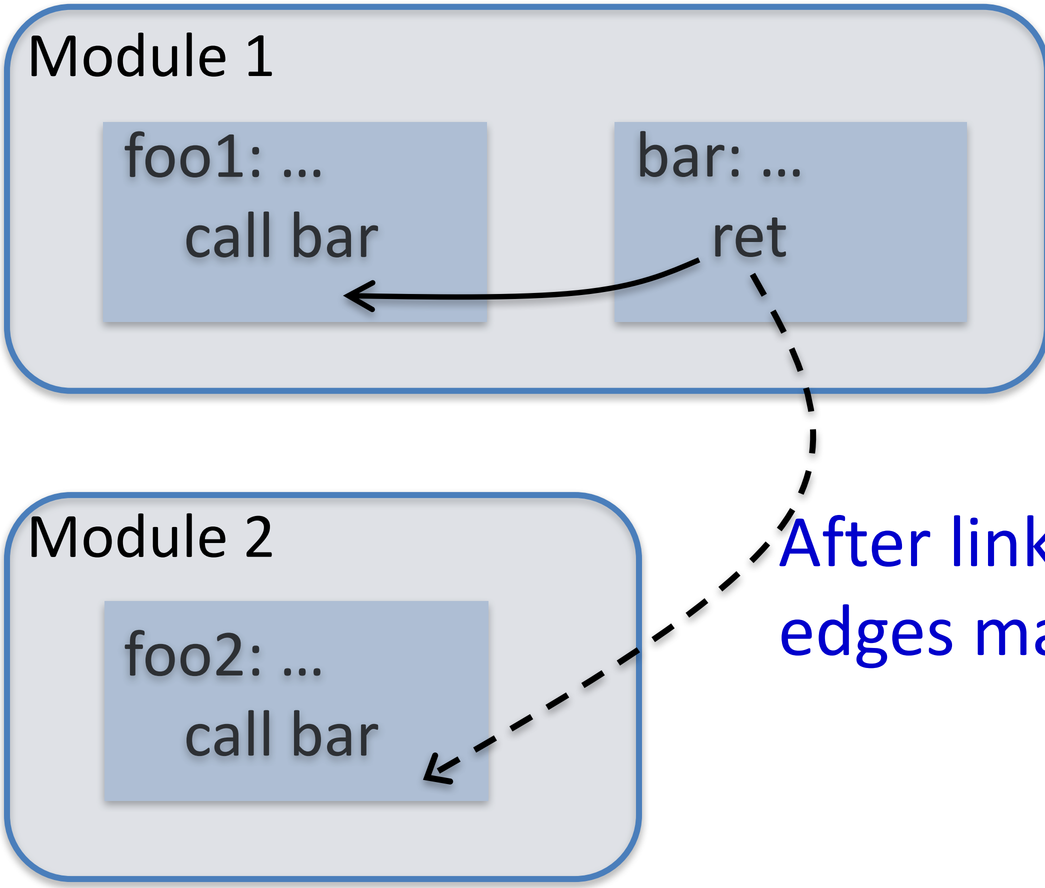
Return to libc

Return-Oriented Programming (ROP) attacks

Previous CFI Work

- Performance: 20-25% overhead in the original CFI work
- No support for **modularity**
  - All code, including libraries, must be available during static compilation time
  - No support for dynamic libraries (or code generated on the fly by just-in-time compilers)
  - Each program has to have its own instrumented version of libraries

CFG Changes When Linking Modules

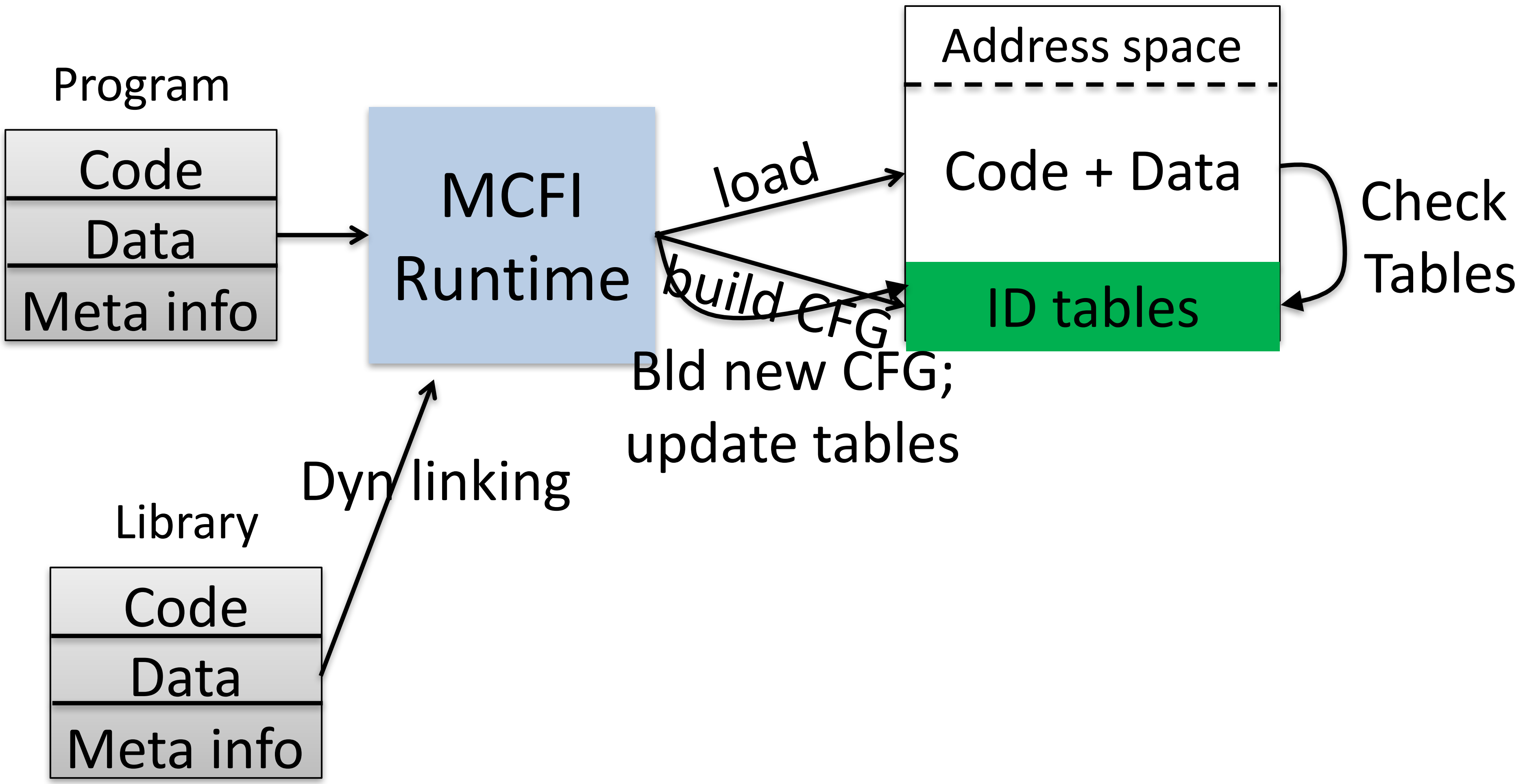


After linking, new edges may be added

Modular Control Flow Integrity (MCFI)
[Niu & Tan PLDI 2014, CCS 2015]

- CFG encoded as centralized tables
  - Checks consult tables for CFI enforcement
  - Updated during dynamic linking
- Benefits of centralized tables
  - Tables separate from code; instrumentation unchanged after tables changed
  - Favorable memory cache effect
  - Easier to achieve thread safety
  - Easier to protect the tables against attacker corruption

MCFI System Flow



CFG Generation for C/C++

- A seemingly easy problem
  - But the hard question is how to compute control-flow edges out of indirect branches
  - Quite complex considering function pointers, signal handlers, virtual method calls, exceptions, etc.
- Tradeoff between precision and performance
  - Remember it has to be performed online when libraries are dynamically linked
  - Sophisticated pointer analysis is perhaps too costly

MCFI's Approach for CFG Generation

- A type-based approach for C/C++ code
- An MCFI module contains code, data, and **meta information** (mostly about types)
- MCFI modules are generated from source code by an augmented LLVM compiler

- Note: there are alternative approaches for CFG generation
  - Dr. Trent Jaeger's group proposed a taint-based approach
  - See posters

CFG Construction for Indirect Branches

- Indirect call "call fp", where fp is of type t*

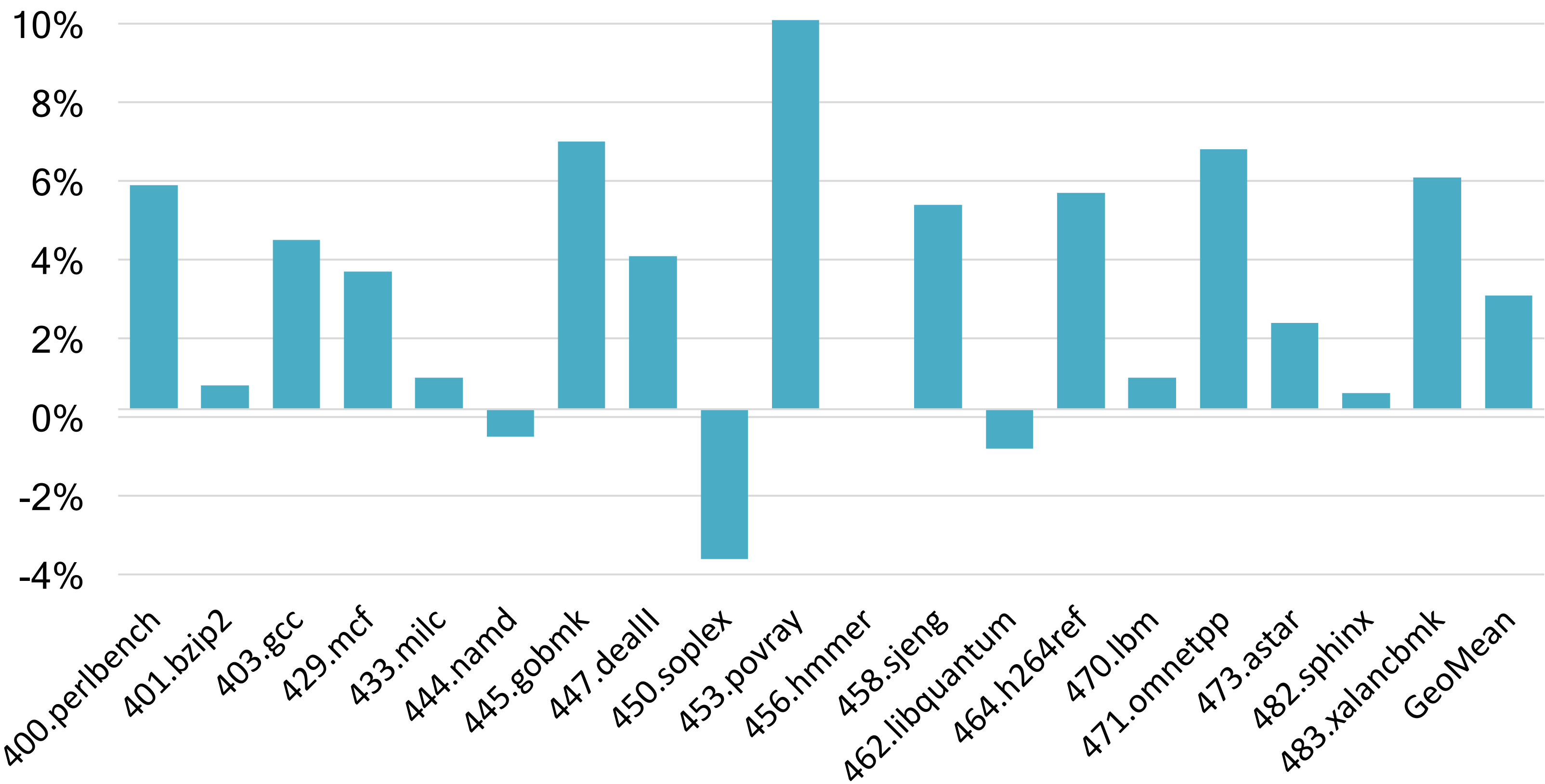
  It is allowed to call function f if

  (1) f's type is some t' that is structurally equivalent to t, and
  (2) f's address is taken in the code (i.e., "&f" is somewhere in code)

- Returns: first construct a call graph; allow a return to go back to any caller in the call graph
  - Also need to take care of tail calls
- Other cases: indirect jumps; setjmp/longjmp, variable-argument functions, signal handlers, …

MCFI Performance Overhead on SPEC2006

On average,2.9%.



Improving the Security of Languages with Managed Runtimes

Languages with Managed Runtimes

Java

JavaScript

php

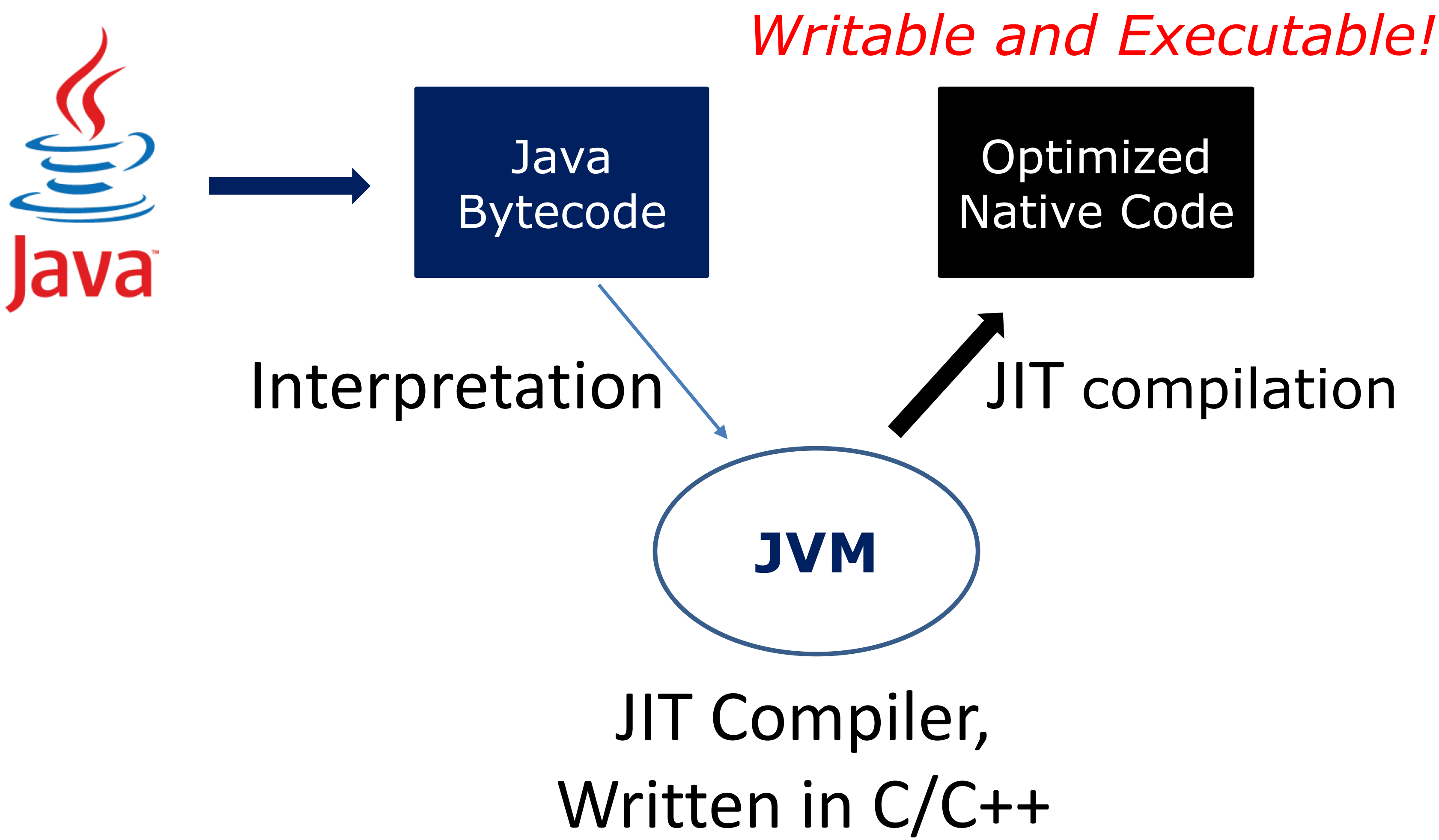
Lua

C#

Managed Runtimes and Security

- A language with a managed runtime is typically safer
  - The runtime restricts program behavior via dynamic monitoring
  - E.g., the Java Virtual Machine performs stack inspection
- However,
  - Managed runtimes are developed in unsafe languages (C++)
  - They use Just-in-Time (JIT) compilation to generate native code on the fly

Performance Boosting Using Just-In-Time Compilation (JIT)



Security Threats to JIT Compilation

- JIT compilers
  - Typically written in C++ for high performance
  - 500,000 to several million lines of code
  - Memory corruption -> control-flow hijacking attacks
- JITted code (native code generated on the fly)
  - JITted code overwriting [Chen et al., 2014]
    - Because the region that contains JITted code is both writable and executable
  - **JIT spraying** [Blazakis, 2010]

JIT Spraying Example

JavaScript code by the attacker

```
var y =
  0x3C0BB090 ^ 0x3C80CD90
```

Normal code execution

```
X86 assembly: movl $0x3C0BB090, %eax; xorl $0x3C80CD90, %eax
Code bytes:   B890B00B3C              3590CD803C
```

If the attacker hijacks the control flow and jumps 1-byte ahead.

```
90   B00B             3C35              90   CD80
nop; movb $0xB, %al; cmpb $0x35, %al; nop; int  $0x80
```

The “exec” system call

Observations

- JIT-spraying is the result of control-flow hijacking
- Modules in JIT compilation
  - The code in a JIT compiler
  - JITted code: dynamically generated code; dynamically linked to the JIT compiler’s code

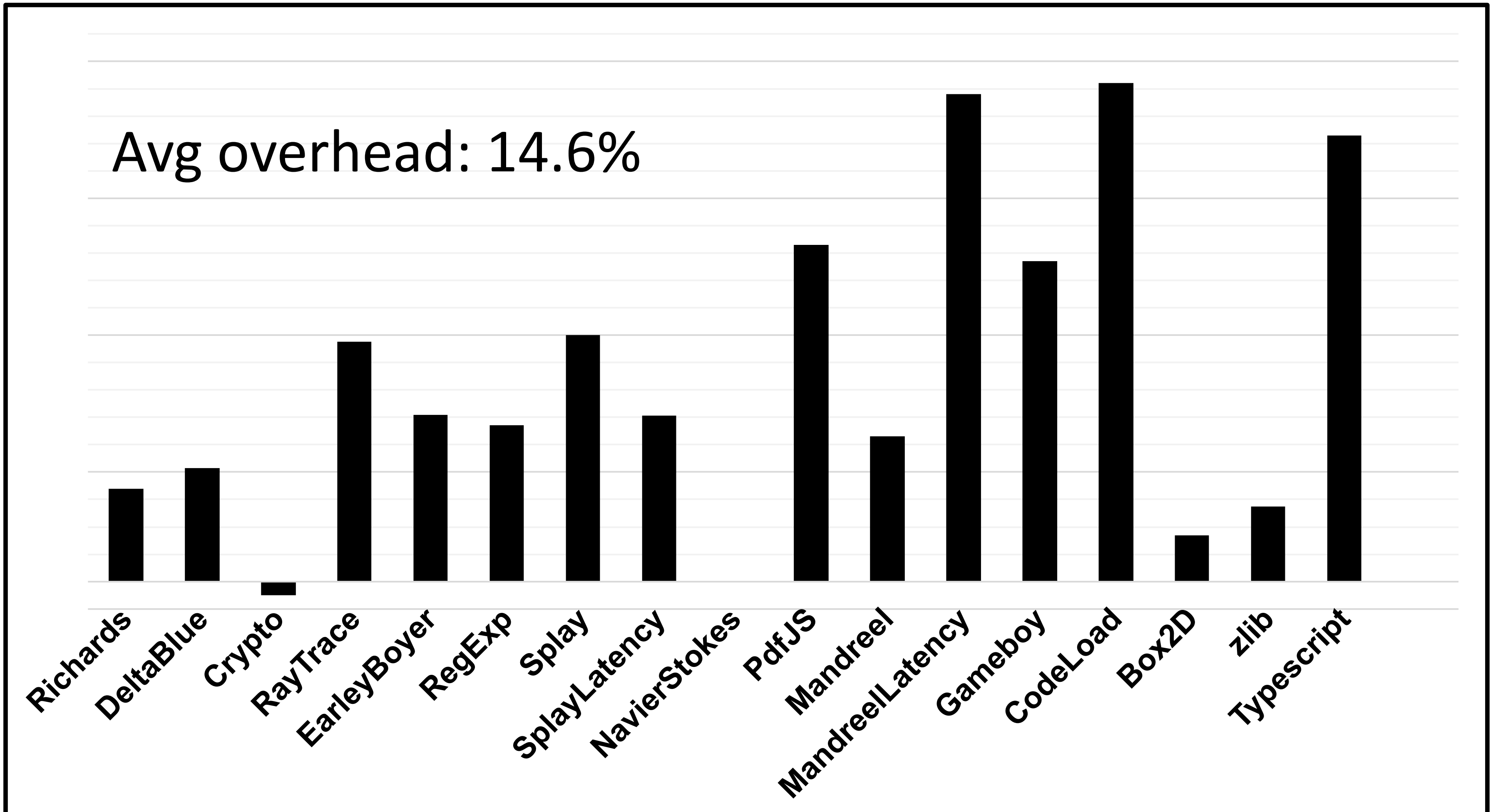
RockJIT [Niu & Tan CCS 2014]

- Extend Modular CFI to cover JIT compilation
- For the JIT compiler
  - (Offline) Statically builds its CFG and encodes it as runtime tables
- JITted code
  - Treat each piece of newly generated code as a new module
  - (Online) Build a new CFG that covers the new code and the JIT compiler's code

Adapting A JIT Compiler to RockJIT

- The code-emission logic needs to be changed to emit MCFI-compatible code (with CFI checks)
- JITted code manipulation should be changed to invoke RockJIT-provided safe primitives
  - **Code installation**: when new code is generated by the JIT compiler
  - **Code modification**: during code optimizations such as inline caching
  - **Code deletion**: when code becomes obsolete
- ~800 lines of source code changes to Google's V8

RockJIT-Protected V8 on Octane 2 JavaScript Benchmarks



Recap

- Compilers can be used to automatically harden code
  - For fault isolation, for control-flow integrity, for …
- To harden dynamic code (dynamic libraries, runtime code generation, …)
  - Some work performed at runtime (e.g., CFG construction)
  - Need to balance security and performance
  - Also need to accommodate concurrency (not discussed)

Some Ongoing Research

- Automatic software partitioning
  - Partitioning monolithic software into least-privileged components
  - Joint with Shen Liu and Dr. Trent Jaeger
- Binary-level reverse engineering and hardening
  - Reverse engineer binary code and perform automatic hardening
  - Joint with Dongrui Zeng
- Compiler-based side channel mitigation
  - Static analysis for side channel identification
  - Program transformation for side channel mitigation
  - Joint with Rob Brotzman-Smith and Dr. Danfeng Zhang
- See posters for details
- …

Acknowledgements

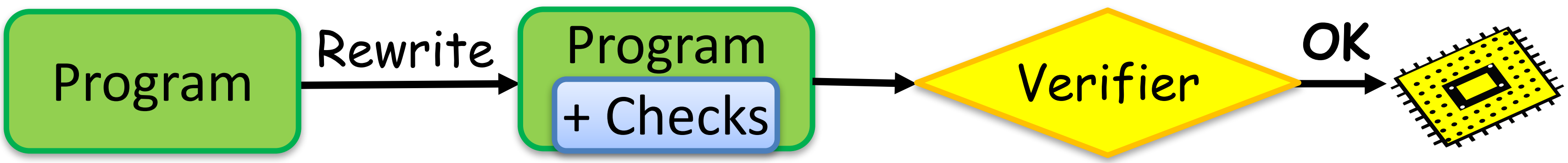
- Sponsors
- Thanks to students and collaborators
  - Students: Ben Niu, Joseph Tassarotti, Edward Gan, Nirupama Talele, Shen Liu, Dongrui Zeng, Rob-Brotzman Smith
  - Collaborators: Greg Morrisett, Trent Jaeger, Danfeng Zhang, Patrick McDaniel, Jean-Baptiste Tristan, Danfeng Yao, Úlfar Erlingsson, Yu David Liu

- MCFI/RockJIT code open sourced: https://github.com/mcfi

Backup slides

Automatic Software Hardening

- Integrate the reference monitor into the code (Inlined Reference Monitors, IRM)



- Verifier: verifying that checks are inlined correctly (so that the proper policy is enforced)
- Benefits
  - Small trusted computing base
  - Low performance overhead (no context switch)
  - Can enforce any safety policy [Schneider 1998]

A Flavor of the x86 Model

- Syntax
  - NOT: bool -> operand -> instr
- Decoding

```
Definition NOT_p : grammar instr :=
  "1111" $$ "011" $$ anybit $ ext_op_modrm2 "010" @
   (fun p => NOT (fst p) (snd p))
```

Decode pattern

Semantic action: construct a NOT instr

A Flavor of the x86 Model, cont'd

- **Semantics**

```
Definition conv_NOT (pre: prefix) (w: bool) (op: operand) : Conv
unit :=
  let load := load_op pre w in
  let set := set_op pre w in
  let seg := get_segment_op pre DS op in
    p0 <- load seg op;
    max_unsigned <- load_Z _ (max_unsigned size32);
    p1 <- arith xor_op p0 max_unsigned;
    set seg p1 op.
```

A Flavor of the Proofs

Lemma NOT_same_pc: forall pre w op,
  same_pc (conv_NOT pre w op).
Proof.
  …
Qed.

NOT does not change the program counter.

Theorem rocksalt_correct: forall …, …
Proof.
  …
Qed.

CFG Statistics for SPEC2006 Programs

SPEC2006	IBs	IBTs	EQCs
perlbench	3327	18378	1857
bzip2	1711	4064	1171
gcc	6108	50412	3258
mcf	1625	3851	1140
gobmk	3908	14556	1631
hmmer	2038	7906	1471
sjeng	1777	4826	1220
libquantum	1688	4169	1182
h264	2455	7046	1526
milc	1825	5879	1310
lbm	1612	3839	1128
sphinx	1893	6431	1369
namd	4795	17552	2829
dealII	13623	61392	7836
soplex	6304	22350	3499
povray	6274	28666	3704
omnetpp	7790	35689	4035
astar	4769	16695	2859
xalancbmk	31166	97186	11281

IBs: # of indirect branches

IBTs: # of possible indirect branch targets

EQCs: # of equivalence classes; upper bounded by IBs

ID Tables

- ID tables encode a CFG
- Divide target addresses into equivalent classes, each assigned an ID
- Branch ID table (Bary table)
  - A map from the location of an indirect branch to the ID of the equivalent class that the indirect branch is allowed to jump to
- Target ID table (Tary table)
  - A map from an address to the ID of the equivalent class of the address
- Conceptually, for an indirect branch,
  - Load the branch ID using the address where the branch is
  - Load the target ID using the real target address
  - Compare the two IDs; if not the same, CFI violation

Thread Safety of Tables

- The tables are global data shared by multiple threads
  - One thread may read the tables to decide whether an indirect branch is allowed
  - Another thread loads a library and triggers an update of the tables
- To avoid data races, wrap table operations into transactions and use Software Transactional Memory (STM)
  - **Check transaction (TxCheck):** used before an indirect branch
  - **Update transaction (TxUpdate):** used when a library is dynamically linked

Why STM?

- A check transaction
  - Performs speculative table reads, assuming no threads are updating the tables
  - If the assumption is wrong, it aborts and retries
- Why is this more efficient than, say, locking?
  - Many more indirect branches compared to loading libraries?
  - **Many more check transactions than update transactions**
  - So check transactions rarely fail