Protecting Dynamic Code by Modular Control-Flow Integrity

Gang Tan

Department of CSE, Penn State Univ.

At International Workshop on Modularity Across the System Stack (MASS)

Mar 14th, 2016, Malaga, Spain
Cyber Insecurity

Hackers Use Heartbleed Bug to Attack 'Major Corporation'

Web attacks build on Shellshock bug

New 'POODLE' Bug Takes Bite Out of SSL 3.0 Web Encryption Protocol
Blame the Software

- Malicious software
- **Buggy software can be as harmful**
  - Benign code with programming mistakes
  - Attackers exploit those mistakes to cause havoc
  - Example: OpenSSL’s Heartbleed bug

OpenSSL

Heartbleed bug

Tiny programming mistakes can cause huge havoc!

Research Question: **automation** to mitigate tiny security-critical programming mistakes?
Compilers to the Rescue

- Compilers for bug finding (perform program analysis)
- Use compilers for **bug toleration**
  - Assume source code is buggy
  - Perform **program transformation** to embed security checks into the executable code
  - Detect attacks during runtime (e.g., StackGuard)
  - AKA Inlined Reference Monitors (IRMs)
What Checks to Insert?

• Ideally, we want to insert checks so that
  – They enforce a well-defined security policy
  – They can catch a large amount of software attacks
  – Runtime slowdown is tolerable

• This talk: control-flow integrity
  – Prevent control-flow hijacking attacks
Control-Flow Hijacking and Control-Flow Integrity
Memory Corruption Errors

- Software written in unsafe languages (C/C++) may suffer from **memory-corruption errors**
  - Buffer overflows (on the stack or on the heap)
  - Use after free bugs; i.e., using some memory after it has been freed
  - Format-string errors
  - ...
Modelling Memory Corruption

• **Threat model**
  – Attacker controls data memory
  – Can corrupt data memory between any two instructions
    • Attacker as a concurrent thread
  – However,
    • Separation between code and data memory
    • Attacker cannot directly change code mem and registers
From Memory Corruption to Control-Flow Hijacking

- Attacker control data memory
  - Code pointers (e.g., return addresses) also in data memory
- Control-flow hijacking
  - Corrupt a code pointer and hijack it to change the control flow
  - A common step in most software attacks
Example of Control-Flow Hijacking

foo: ...
call bar

bar: ...
ret

Injected code

A library function

Code gadgets

What if bar has a buffer overflow and the return address is hijacked?

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
   • Indirect branches include returns, indirect calls, and indirect jumps
   • Instrumentation: insert checks before indirect branches

CFI Policy: execution of the instrumented program follows a 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

- **foo**: ... call bar
- **bar**: ...
  - ret
  - CFI-ret

- Injected code
- A libc function
- Code gadgets

- Stack smashing
- Return to libc
- Return-Oriented Programming (ROP) attacks

Check if the target is allowed by the CFG
CFI Instrumentation Steps

• For each indirect branch
  – CFG tells the set of possible targets; use an ID for this equivalence class of targets
  – Insert an ID-encoding no-op at every target
  – Insert an ID-check instruction before the indirect branch

Target 1

```
foo1: ...
call bar
no-op(ID)
```

Target 2

```
foo2: ...
call bar
no-op(ID)
```

Target 1

```
bar: ...
check(ID)
ret
```
Why Not Just Safe Languages?

- Using safe languages (e.g., Java, JavaScript, ...) improves software security substantially
  - Use safe languages as much as we can
- On the other hand,
  - **Performance**: 2-10x slowdown when using safe languages
  - **Legacy code**: a lot of mature libraries in C/C++
  - **Big language runtimes** for safe languages
    - E.g., a typical just-in-time (JIT) engine for JavaScript has at least 500,000 lines of code written in C++
    - Attacks on language runtimes are already in the wild: JIT-spraying attacks
Extending CFI with Modularity
Classic CFI Lacks Modularity

- The construction of CFG
  - Typically requires a global analysis
- The inserted IDs cannot overlap with the rest of the code
  - Cannot guarantee it without access to all the code
- As a result
  - All code, including libraries, must be available during instrumentation time
  - Each program has to have its own instrumented version of libraries
  - No support for separate compilation and dynamic linking
  - The biggest obstacle to CFI’s practicality
CFG Changes When Linking Modules

Module 1
- foo1: ...
  call bar
- bar: ...
  ret

Module 2
- foo2: ...
  call bar

After linking, new edges may be added
Modular Control Flow Integrity (MCFI) [Niu & Tan PLDI 2014]

- CFG encoded as centralized tables
  - Consult information in tables for CFI enforcement
  - During dynamic linking, compute new CFG and update tables
  - Type-based CFG generation

- Benefits of using 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

Program
- Code
- Data
- Meta info

MCFI Runtime

Address space
- Code + Data
- ID tables

Library
- Code
- Data
- Meta info

Dyn linking
- load
- build CFG
- Bld new CFG; update tables

Check Tables
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
CFG Construction for Indirect Branches

- Indirect calls: an indirect call through a function pointer of type $t^*$ is allowed to call any function if (1) the function’s type is some $t'$ that is structurally equivalent to $t$, and (2) the function’s address is taken in the code.

- Returns: first construct the 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, ...
### CFG Statistics for SPEC2006 Programs

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

**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
MCFI Performance Overhead on SPEC2006

On average, 2.9%. 
Use Modular CFI to Improve the Security of JIT Compilation
Languages with Managed Runtimes

- Java
- JavaScript
- PHP
- Lua
- C#
Performance Boosting Using Just-In-Time Compilation (JIT)

Java Bytecode

Interpretation

JIT compilation

JIT Compiler Written in C/C++

Writable and Executable!
Security Threats to JIT Compilation

• JIT compilers
  – 500,000 to several million lines of code
  – Typically written in C++ for high performance
  – 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

```javascript
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.

```x86 assembly
nop; movb $0xB, %al; cmpb $0x35, %al; nop;
```

```
in %0x80
```

The “exec” system call
Observations

• JIT-spraying on JIT 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 ID 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

Avg: 14.6%
A Brief Recap

• To accommodate dynamic code
  – Do most of the work online
  – MCFI’s runtime: construct the CFG; build tables; …

• Sacrifices when going online
  – Have to opt for fast, simple analysis
  – MCFI: type-based CFG generation
  – CFG precision may suffer (compared to an approach that uses sophisticated pointer analysis)

• However, it’s not a one-sided story
  – Dynamic analysis can help improve CFG precision
PICFI: Enforcing Per-Input CFG
CFG Precision and Security

- CFI’s security policy is its enforced CFG
- A CFG is an **over-approximation** of a program’s runtime control flow
  - A program can have many CFGs
- Even after a CFG is enforced,
  - Attacker is allowed to change a program’s control flow within the CFG
  - The more tight a CFG is, less wiggle room an attacker has
- Recent attacks on CFI of various precisions
  - Attacks on certain programs with fine-grained CFI: [Carlini et al. Usenix Security 2015]; ...
All-Input CFG versus Per-Input CFG

- Past CFI: enforce a CFG considering all possible program inputs
- The CFG for a particular input can be more precise (better security)
Per-Input CFI (PICFI or πCFI) [Niu and Tan CCS 2015]

• The goal is to enforce a per-input CFG
  – However, impossible to compute and store a CFG for each input
• Idea: lazy edge addition
  – Start with the empty CFG (just nodes, but no edges)
  – At runtime, before an edge is needed, add the edge to the CFG

Suppose input is 0
Making it Secure

• Cannot allow program to add arbitrary edges
  – First build an all-input CFG ahead of time
  – Only allow edges in the all-input CFG added to the per-input CFG

• Per-input CFG
  – Empty at the beginning
  – It grows monotonically, but upper-bounded by all-input CFG
  – The hope is that per-input CFG has less edges than all-input CFG and thus provides stronger constraints on legal control flow
Making it Efficient

• Edge addition is costly
• Instead, address activation
  – When an edge is needed, activate the edge’s target address: all edges targeting the address are added to the per-input CFG
  – Cons: less precise compared to edge addition
  – Pro: each address is activated at most once
Address Activation For Return Addresses

foo: ...
activate(addr)
call bar
addr: →

bar: ...
ret
Performance Overhead on SPEC2006

On average, 3.2% for πCFI, 0.3% more than MCFI.
Per-Input CFG Statistics

<table>
<thead>
<tr>
<th>SPECCPU2006</th>
<th>Indirect branch targets activated (%)</th>
<th>Indirect branch edges activated (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>22.5%</td>
<td>15.4%</td>
</tr>
<tr>
<td>403.gcc</td>
<td>28.6%</td>
<td>6.1%</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>25.3%</td>
<td>13.9%</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>21.4%</td>
<td>13.5%</td>
</tr>
</tbody>
</table>

About <30% of indirect branch targets are activated compared to the all-input CFG.

Reason: applications contain code for error handling, for processing different configurations; all-input CFG computation has to over-approximate; ...
What’s Learned and Future Work
What’s Learned

• Modularity has many aspects
  – Writing code modularly (e.g., AOP)
  – Separate compilation
  – Modular reasoning about program properties
    • E.g., CFG construction
  – Accommodating dynamic code
    • Code that is not statically available: dynamic libraries; code generated on the fly; self modification

• Our way of handling modularity
  – Ask compilers include metadata in object code
  – Modular reasoning at runtime (during library loading and code generation)
  – Can perform dynamic analysis to reap some benefits (e.g. PICFI)
What’s Learned

• Different requirements from typical dynamic analysis
  – Typical dynamic analysis: use traces for bug finding, for debugging concurrent code, ...
    • It’s okay if it’s slow
  – In our setting, analysis performed adds to the program’s execution time
    • Cannot tolerate slow analysis
    • In security, at most 5 to 10% slowdown
  – **Wanted**: fast, modular points-to analysis for more accurate CFG construction
What’s Learned

• Often multithreading in security monitoring is a tricky issue
  – Need concurrent data structures to store metadata
    • E.g., our ID tables
  – Efficient and thread safe
  – **Wanted**: hardware support would be nice; for example, an tagged architecture
Future Work on CFI

• Formalization
  – CFI in the presence of dynamic linking and JITting

• Relation between security and CFG precision
  – How to qualify/quantify the security gains of when CFG is more precise?

• Context-sensitive CFI

• OS-level CFI support
  – Microsoft’s Control-Flow Guard is a good start, but too coarse grained

• ...

Acknowledgements

- Support from NSF, Google Research, IAI incorporated
- Actual work done by Ben Niu for his PhD thesis “Practical Control-Flow Integrity”
- Code open sourced: https://github.com/mcfi
Conclusions

• CFI is fundamental to software security
  – Detect control-flow deviations
  – The basis for other inlined reference monitors

• MCFI enhances security and incurs low performance overhead
  – Overhead comparable to existing coarse-grained CFI

• MCFI makes CFI practical by supporting modularity

• Hopefully it can be adopted to support a more secure world
  – FreeBSD follow up