Control-Flow Integrity Principles, Implementations, and Applications

Martn Abadi  
Computer Science Dept  
University of California  
Santa Cruz

Mihai Budiu  
Microsoft Research  
Silicon Valley

Ulfar Erlingsson  
Microsoft Research  
Silicon Valley

Jay Ligatti  
Dept. of Computer Science  
Pricenton University  
Computer Science Department

April 8, 2019
Computers are subject to external attacks. The aim is to trigger a pre-existing software flaw. By exploiting such flaws, the attacker may gain control over software behavior.
Computers are subject to external attacks
Problem

- Computers are subject to external attacks
- Aim is to trigger a pre-existing software flaw
Background

Problem

- Computers are subject to external attacks
- Aim is to trigger a pre-existing software flaw
- By exploiting such flaws, the attacker may gain control over software behavior
Contributions of this paper

Describe studies and one mitigation technique (CFI)

Introduce CFI enforcement

Present an implementation for Windows on x86 architecture
Contributions of this paper

- Describe studies and one mitigation technique (CFI)
Contributions of this paper

- Describe studies and one mitigation technique (CFI)
- Introduce CFI enforcement
Contributions of this paper

- Describe studies and one mitigation technique (CFI)
- Introduce CFI enforcement
- Present an implementation for Windows on x86 architecture
Control Flow Integrity - CFI
Control Flow Integrity - CFI

- Mitigation technique which prevents a wide variety of attacks (code-reuse attacks)
Control Flow Integrity - CFI

- Mitigation technique which prevents a wide variety of attacks (code-reuse attacks)
- Follows a **Control Flow Graph - CFG** which is determined ahead of time
Control Flow Graph - CFG

Figure: if-then-else control flow graph

This paper focuses on Static Binary Analysis.

Dimitris Karnikis

CS-558 INTERNET SYSTEMS & TECHNOLOGIES
Control Flow Graph - CFG

- A graph that describes the flow of a program
Control Flow Graph - CFG

- A graph that describes the flow of a program
- Can be defined by:

Figure: if-then-else control flow graph
Control Flow Graph - CFG

- A graph that describes the flow of a program
- Can be defined by:
  - Source code analysis

Figure: if-then-else control flow graph
Control Flow Graph - CFG

- A graph that describes the flow of a program
- Can be defined by:
  1. Source code analysis
  2. Static Binary analysis
Background

Control Flow Graph - CFG

- A graph that describes the flow of a program
- Can be defined by:
  1. Source code analysis
  2. Static Binary analysis
  3. Execution profiling

Figure: if-then-else control flow graph

This paper focuses on Static Binary Analysis
Control Flow Graph - CFG

- A graph that describes the flow of a program
- Can be defined by:
  1. Source code analysis
  2. Static Binary analysis
  3. Execution profiling

Figure: if-then-else control flow graph
**Control Flow Graph - CFG**

- A graph that describes the flow of a program
- Can be defined by:
  1. Source code analysis
  2. Static Binary analysis
  3. Execution profiling

*Figure: if-then-else control flow graph*

This paper focuses on Static Binary Analysis
Stack Canaries

Canaries or canary words are known values placed between a buffer and control data on the stack to monitor for potential buffer overflows.
Stack Canaries

- Canaries or canary words are known values
Stack Canaries

- Canaries or canary words are known values
- Such values are placed between a buffer and control data on the stack to monitor for potential buffer overflows
Stack Canaries

- Canaries or canary words are known values.
- Such values are placed between a buffer and control data on the stack to monitor for potential buffer overflows.
Inlined Reference Monitor - IRM
Inlined Reference Monitor - IRM

- Is a concept that defined a set of design requirements on a reference validation mechanism
Shadow Stack
Shadow Stack

- Mechanism that protects a procedure’s stored return address
Shadow Stack

- Mechanism that protects a procedure’s stored return address
Mitigation Techniques
Mitigation Techniques

- Stack canaries
Mitigation Techniques

- Stack canaries
- Runtime elimination of buffer overflows
Mitigation Techniques

- Stack canaries
- Runtime elimination of buffer overflows
- Randomization and artificial heterogeneity
They examined many concrete attacks and they have found that CFI enforcement prevents most of them such as:

- stack-based overflow attacks
- heap-based jump-to-libc attacks

but CFI is not panacea
They examined many concrete attacks and they have found that CFI enforcement prevents most of them such as:

- stack-based overflow attacks
They examined many concrete attacks and they have found that CFI enforcement prevents most of them such as:

- stack-based overflow attacks
- heap-based jump-to-libc attacks
They examined many concrete attacks and they have found that CFI enforcement prevents most of them such as:

- stack-based overflow attacks
- heap-based jump-to-libc attacks

but
They examined many concrete attacks and they have found that CFI enforcement prevents most of them such as:

- stack-based overflow attacks
- heap-based jump-to-libc attacks

but

CFI is not panacea 😞
They examined many concrete attacks and they have found that CFI enforcement prevents most of them such as:

- stack-based overflow attacks
- heap-based jump-to-libc attacks

**but**

CFI is not panacea 😞

Exploits that lie within the bounds of the allowed CFG are not prevented.
CFI can prevent the circumvention of two well-known security enforcement mechanisms.
CFI can prevent the circumvention of two well-known security enforcement mechanisms

⇒ Inlined Reference Monitors (IRMs)
CFI can prevent the circumvention of two well-known security enforcement mechanisms

⇒ Inlined Reference Monitors (IRMs)
⇒ Software Fault Isolation (SFI)
CFI can prevent the circumvention of two well-known security enforcement mechanisms

⇒ Inlined Reference Monitors (IRMs)
⇒ Software Fault Isolation (SFI)

In particular, CFI can
CFI can prevent the circumvention of two well-known security enforcement mechanisms

⇒ Inlined Reference Monitors (IRMs)
⇒ Software Fault Isolation (SFI)

In particular, CFI can

⇒ help to protect security data such as shadow call stack
Software Fault Isolation and Inlined Reference Monitors

- SFI is a special IRM
Software Fault Isolation and Inlined Reference Monitors

- SFI is a special IRM
- SFI performs dynamic checks on memory protection
Software Fault Isolation and Inlined Reference Monitors

- SFI is a special IRM
- SFI performs dynamic checks on memory protection

IRM implementations must consider
SFI is a special IRM

SFI performs dynamic checks on memory protection

IRM implementations must consider

⇒ that a subject program may attempt to bypass the added checks
Software Fault Isolation and Inlined Reference Monitors

- SFI is a special IRM
- SFI performs dynamic checks on memory protection

IRM implementations must consider

⇒ that a subject program may attempt to **bypass** the added checks
⇒ solutions that **impose restrictions** on control flow
Those difficulties exist due to x86 variable-length sequences of opcodes for instructions.
Software Fault Isolation and Inlined Reference Monitors

- Those difficulties exist due to x86 variable-length sequences of opcodes for instructions.
- IRMs are problematic because the need control flow checks.
Software Fault Isolation and Inlined Reference Monitors

- Those difficulties exist due to x86 variable-length sequences of opcodes for instructions.
- IRMs are problematic because the need control flow checks.

So CFI may serve as the foundation for efficient IRM implementations.
Inlined CFI Enforcement

Enforced CFI by Instrumentation

When machine code transfers control (e.g. jumps)
Enforcing CFI by Instrumentation

When machine code transfers control (e.g. jumps)

1. Target must be a valid destination determined by CFG
### Enforcing CFI by Instrumentation

When machine code transfers control (e.g. jumps):

1. Target must be a **valid** destination determined by CFG
2. But for targets that are computed at runtime, dynamic check is needed
Inlined CFI Enforcement

Enforcing CFI by Instrumentation

When machine code transfers control (e.g. jumps)

1. Target must be a **valid** destination determined by CFG
2. But for targets that are computed at runtime, dynamic check is needed

⇒ New machine-code instruction with an operand **ID**
Enforcing CFI by Instrumentation

When machine code transfers control (e.g. jumps):

1. Target must be a **valid** destination determined by CFG
2. But for targets that are computed at runtime, dynamic check is needed
   - New machine-code instruction with an operand **ID**
   - A call instruction **call ID, DST** providing that DST code starts with ID
Enforcing CFI by Instrumentation

When machine code transfers control (e.g. jumps)

1. Target must be a **valid** destination determined by CFG
2. But for targets that are computed at runtime, dynamic check is needed
   - New machine-code instruction with an operand **ID**
   - A call instruction **call ID, DST** providing that DST code starts with ID
   - A corresponding **ret ID**
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len) {
    sort( a, len, lt );
    sort( b, len, gt );
}

Figure 1: Example program fragment and an outline of its CFG and CFI instrumentation.
### Inlined CFI Enforcement

#### Source Instructions

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Instructions</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF E1</td>
<td>jmp ecx</td>
<td>; computed jump</td>
</tr>
</tbody>
</table>

#### Destination Instructions

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Instructions</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>8B 44 24 04</td>
<td>mov eax, [esp+4]</td>
<td>; dst</td>
</tr>
</tbody>
</table>

... 

can be instrumented as (a):

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Instructions</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>81 39 78 56 34 12</td>
<td>cmp [ecx], 12345678h</td>
<td>; comp ID &amp; dst</td>
</tr>
<tr>
<td>75 13</td>
<td>jne error_label</td>
<td>; if != fail</td>
</tr>
<tr>
<td>8D 49 04</td>
<td>lea ecx, [ecx+4]</td>
<td>; skip ID at dst</td>
</tr>
<tr>
<td>FF E1</td>
<td>jmp ecx</td>
<td>; jump to dst</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Instructions</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>78 56 34 12</td>
<td></td>
<td>; data 12345678h</td>
</tr>
</tbody>
</table>

or, alternatively, instrumented as (b):

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Instructions</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>B8 77 56 34 12</td>
<td>mov eax, 12345677h</td>
<td>; load ID-1</td>
</tr>
<tr>
<td>40</td>
<td>inc eax</td>
<td>; add 1 for ID</td>
</tr>
<tr>
<td>39 41 04</td>
<td>cmp [ecx+4], eax</td>
<td>; compare w/dst</td>
</tr>
<tr>
<td>75 13</td>
<td>jne error_label</td>
<td>; if != fail</td>
</tr>
<tr>
<td>FF E1</td>
<td>jmp ecx</td>
<td>; jump to label</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Instructions</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>3E 0F 18 05</td>
<td>prefetchnta</td>
<td>; label</td>
</tr>
<tr>
<td>78 56 34 12</td>
<td></td>
<td>[12345678h]</td>
</tr>
<tr>
<td>8B 44 24 04</td>
<td>mov eax, [esp+4]</td>
<td>; dst</td>
</tr>
</tbody>
</table>
Assumptions

1. **UNQ** - Unique IDs: The patterns chosen as IDs must be *unique*
Assumptions

1. **UNQ** - Unique IDs: The patterns chosen as IDs must be unique

2. **NWC** - Non-Writable Code: It must not be possible for the program to modify code memory
Assumptions

1. **UNQ** - Unique IDs: The patterns chosen as IDs must be unique

2. **NWC** - Non-Writable Code: It must not be possible for the program to modify code memory

3. **NXD** - Non-Executable Data: It must not be possible to execute data as if it were code
Based on **Vulkan**

- State-of-art instrumentation system for x86 binaries
Based on **Vulkan**

- State-of-art instrumentation system for x86 binaries
- Requires neither recompilation nor source-code access
Based on **Vulkan**

- State-of-art instrumentation system for x86 binaries
- Requires neither recompilation nor source-code access
- Implemented on Windows
<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Function Call</th>
<th>Instructions</th>
<th>Function Return</th>
<th>Opcode bytes</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF 53 08</td>
<td>call [ebx+8] ; call fptra</td>
<td></td>
<td></td>
<td>C2 10 00</td>
<td>ret 10h</td>
</tr>
</tbody>
</table>

are instrumented using prefetchnta destination IDs, to become

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Instruction</th>
<th>Instructions</th>
<th>Opcode bytes</th>
<th>Instruction</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>8B 43 08</td>
<td>mov eax, [ebx+8] ; load fptra</td>
<td></td>
<td>8B 0C 24</td>
<td>mov ecx, [esp] ; load ret</td>
<td></td>
</tr>
<tr>
<td>3E 81 78 04 78 56 34 12</td>
<td>cmp [eax+4], 12345678h ; comp w/ID</td>
<td></td>
<td>83 C4 14</td>
<td>add esp, 14h ; pop 20</td>
<td></td>
</tr>
<tr>
<td>75 13</td>
<td>jne error_label ; if ! = fail</td>
<td></td>
<td>3E 81 79 04</td>
<td>cmp [ecx+4], ; compare</td>
<td></td>
</tr>
<tr>
<td>FF D0</td>
<td>call eax ; call fptra</td>
<td></td>
<td>DD CC BB AA</td>
<td>AABBCDDh ; w/ID</td>
<td></td>
</tr>
<tr>
<td>3E 0F 18 05 DD CC BB AA</td>
<td>prefetchnta [AABBCDDh] ; label ID</td>
<td></td>
<td>75 13</td>
<td>jne error_label ; if!=fail</td>
<td></td>
</tr>
<tr>
<td>FF E1</td>
<td>jmp ecx ; jump ret</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3:** The CFI instrumentation of x86 call and ret used in our implementation.
Conclusion

Figure 4: Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

Figure 8: Enforcement overhead for CFI with a protected shadow call stack on SPEC2000 benchmarks.
Thank You all!
Questions?