Adaptive Call-site Sensitive Control Flow Integrity

Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou †, Yueqiang Cheng ‡

Dept. of Computer Science, Florida State University, Tallahassee, USA
† School of Computer Science, Zhejiang University, Hangzhou, China
‡ Baidu X-lab, Sunnyvale, USA
Control Flow Integrity

Control Flow Integrity (CFI) is a defense mechanism against control-flow hijacking that employs inline reference monitor to enforce the run-time control flow of a process must follow the statically computed control-flow graph (CFG).

CFI consists of:

- CFI Policy
- Inline Reference Monitor
- CFG
CFI Policy

- Context-insensitive (CI-) CFI: CFI policy without additional information
- Context-sensitive (CS-) CFI: CFI policy with past execution history
  - e.g., path sensitivity
**CFI Metrics**

- **Equivalence Class (EC):** A group of targets that CFI cannot distinguish.
- **Largest EC (LC):** Largest allowed targets among all EC’s.

**CI-CFI**
- # of EC = 1 (D)
- EC Size = 2 (D->E,F)
- LC = 2 (D->E,F)

**CS-CFI**
- # of EC = 2 ({B,D},{C,D})
- EC Size = 1 ({B,D}->E, {C,D}->F)
- LC = 1 ({B,D}->E, {C,D}->F)
Quantifying CFI

To quantify security guarantee of CFI, we propose:

\[ QS_{CFI} = AVG_{EC} \times LC \]

Benefits:

- \( QS_{CFI} \) has a theoretical limit of 1, i.e., every target can be individually distinguished and validated.
- Applicable to both context-sensitive and insensitive CFI systems
  - \( AVG_{EC} \) avoids the false impression of improvements because context-sensitive CFI can exponentially increase the # of ECs
Call-site Sensitive CFI

At runtime, combine backward edge information (return addresses) with intended target at indirect forward edge call-point:

- By accessing call stack (protected by shadow stack, Intel CET etc.).
- Validates multi-level call-sites against a pre-computed CFG.
- Benefits:
  - Strict CFI policy
  - No extra memory
  - No specialized hardware required
  - Whole-program protection

1. \{L29,L5\} => on_admin
2. \{L31,L5\} => on_failure
Challenges

- **Context-sensitivity is expensive.**
  - Runtime reference monitor:
    - Complex verification method.
    - Extra instrumentation required for integrity.

- **Context-sensitivity is complicated.**
  - Unavailable scalable static points-to analysis to compute CFG:
    - Precision.
    - Completeness.
CFI-LB: Control Flow Integrity with Look Back

● Strong CFI Policy:
  ○ Call-site sensitivity.
  ○ SafeStack for secure call stack.

● Fast and Secure CFI Enforcement:
  ○ Adaptive call-site depth.
  ○ Hash-table based set membership test.
  ○ Intel TSX for guaranteed atomicity.

● Complete and precise CFG:
  ○ Multi-scope CFG
    ■ Dynamic-concolic CFG (CS-CFI)
    ■ Static CFG (CI-CFG)
  ○ Localized concolic execution
Adaptive Call-site Sensitivity

- Individual call-point has independent level of call-site sensitivity.
  - TABLE I: 149 call-points have best security guarantee without any context.
  - TABLE II: LC of 403.gcc is call-site insensitive.

<table>
<thead>
<tr>
<th>Sizes</th>
<th>Call-site(0)</th>
<th>Call-site(1)</th>
<th>Call-site(2)</th>
<th>Call-site(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>149</td>
<td>674</td>
<td>2372</td>
<td>5423</td>
</tr>
<tr>
<td>2</td>
<td>27</td>
<td>54</td>
<td>223</td>
<td>667</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>18</td>
<td>48</td>
<td>172</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>17</td>
<td>35</td>
<td>70</td>
</tr>
<tr>
<td>5-10</td>
<td>20</td>
<td>26</td>
<td>56</td>
<td>93</td>
</tr>
<tr>
<td>11-20</td>
<td>4</td>
<td>4</td>
<td>28</td>
<td>37</td>
</tr>
<tr>
<td>21-40</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>54</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>220</td>
<td>795</td>
<td>2763</td>
<td>6463</td>
</tr>
</tbody>
</table>

TABLE I: EC distribution over sizes of the target sets (403.gcc)

<table>
<thead>
<tr>
<th>Call-site Depth</th>
<th>Max. Target Set Size</th>
<th># of Indirect Calls</th>
<th># of ECs</th>
<th>Avg. EC Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>54</td>
<td>161</td>
<td>161</td>
<td>1.48</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>23</td>
<td>262</td>
<td>1.37</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>36</td>
<td>787</td>
<td>1.5</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td>1.47</td>
</tr>
</tbody>
</table>

TABLE II: Distribution of adaptive call-site sensitivity (403.gcc)
Atomicity of Reference Monitor

- Complex monitoring system may cause internal states spill to stack.
- In the multi-threaded programming environment, race condition is critical.
- Solution:
  - Encapsulates reference monitor with Intel TSX hardware transactional memory protection.
    - CFI-LB uses restricted transactional memory (RTM) interface.
  - If a race condition is detected, CFI-LB retries the execution.
    - Transaction could fail without race condition because of cache conflicts.
Multi-scope CFG

● Why?
  ○ Unavailability of scalable context-sensitive static points-to analysis.
  ○ Completeness is crucial.
  ○ Imprecision is unacceptable.
● Dynamic CS-CFG
  ○ Seed input derived.
● Concolic CS-CFG
  ○ Localized concolic execution.
● Static CI-CFG
  ○ Act as a fail-safe for dynamic-concolic CFG.
  ○ In release binary, a control transfer validated by static CFG will be logged.
Dynamic CFG

- Due to limited seed input:
  - Dynamic execution can cover limited number of paths.

- Possible solution:
  - Symbolic execution based complete path exploration.
  - Issues with symbolic execution:
    - Constraint solver is slow.
    - Path explosion problem.
Localized Concolic Execution

- For $n$ call-site sensitivity:
  - Coverage starts from every function in $n$ call-site depth for each indirect call-point.
  - Localized code coverage: all-possible path reachable to an indirect call-point.
  - Intuition: Call-site sensitive code pointers must have assigned within localized coverage.
- Reconstruct memory layout:
  - Utilize concrete execution memory dump (Concolic).
  - Symbolize memories (except references).
    - Global variables.
    - Arguments at coverage entry points.
    - I/O memories in libc function.
- Benefits:
  - Localized all-possible path coverage (*Scalable*).
  - Concolic data-driven path exploration (*Precision*).
Evaluation

We separate our evaluation:

● Effectiveness
  ○ Call-site sensitivity
  ○ Localized concolic CFG

● Performance

● Exploitation
Call-site Sensitive as Context

Deeper context (call-site depth) provide better security.
## Effectiveness of CFI-LB

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Max Level</th>
<th># of Indirect Calls</th>
<th>AVG EC</th>
<th>Final AVG EC</th>
<th>LC</th>
<th>QS&lt;sub&gt;CFI-LB&lt;/sub&gt; /QS&lt;sub&gt;CFI(0)&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>3</td>
<td>62/8/3/7</td>
<td>1.02/1.0/1.21/2.77</td>
<td>2.28</td>
<td>115</td>
<td>1/2.4</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>0</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>403.gcc</td>
<td>2</td>
<td>161/23/36</td>
<td>1.48/1.37/1.50</td>
<td>1.47</td>
<td>54</td>
<td>1/1.84</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>1</td>
<td>36/15/12</td>
<td>16.86/9.44/12.76</td>
<td>12.86</td>
<td>427</td>
<td>1/2.4</td>
</tr>
<tr>
<td>456.hmmr</td>
<td>0</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>2</td>
<td>68/2/4/1</td>
<td>1.5/1.05/1.15/1.25</td>
<td>1.31</td>
<td>2</td>
<td>1/6.4</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>2</td>
<td>226/2/8/3</td>
<td>1.81/1.0/1.04/1.7</td>
<td>1.44</td>
<td>168</td>
<td>1/1.45</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>2</td>
<td>1960/25/30/48</td>
<td>1.06/1.12/1.20/1.71</td>
<td>1.14</td>
<td>26</td>
<td>1/1.52</td>
</tr>
<tr>
<td>444.namd</td>
<td>0</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>447.dealII</td>
<td>2</td>
<td>100/3/4/1</td>
<td>1.04/1.0/1.0/1.11</td>
<td>1.03</td>
<td>2</td>
<td>1/1.07</td>
</tr>
<tr>
<td>450.soplex</td>
<td>0</td>
<td>56</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>453.povray</td>
<td>2</td>
<td>40/3/9</td>
<td>1.6/4.2/2.12</td>
<td>2.12</td>
<td>9</td>
<td>1/1.06</td>
</tr>
<tr>
<td>NGinx</td>
<td>3</td>
<td>94/18/0/11</td>
<td>5.54/1.06/0.0/4.91</td>
<td>3.73</td>
<td>62</td>
<td>1/3.3</td>
</tr>
</tbody>
</table>

TABLE III: Effectiveness of adaptive call-site sensitivity. The table shows the max call-site level, the number of indirect calls in each level, and for each level, the number of ECs and the average EC size. The last column shows the improvement of CFI-LB over context-insensitive CFI.
TABLE IV: Comparing static, dynamic, and concolic CFGs for the Spec CPU2006 benchmarks. Column 2 to 4 show the total number of entries in these CFGs, respectively. Note that number of static-CFG is not directly comparable to these of dyn-CFG and con-CFG because the latter two CFGs have contexts (hence more entries).
Performance Overhead

- Intel core-i7 6700 processor (skylake) with a base frequency of 3.4GHz and 16GB of memory
- Spec CPU2006 benchmark and NGinx
  - 2.7% for the forward-edge protection and 4.8% for the full protection.

Fig: Performance overhead, Average_ex shows the average overhead excluding the three benchmarks that have no overhead.
Exploitation Example

- CVE-2014-1912 (python-2.7.6)
- Root cause: missing check of the buffer size and the receive size in Python’s socket module

```
import socket
r, w = socket.socketpair()
up.send(b'\x90' * 305 + '\x00' + '\x65' + '\x51' + '\x00')
up.recvfrom_into(bytearray(), 305)
```

Fig. 8. PoC exploit for Python CVE-2014-1912

- CFI-LB at exploitable call-point:
  - Use call-site depth 3
  - Allowed 5 valid targets
  - Avg. EC size: 1.0
- CFI-LB can detect any control hijack at this point

```
if (tp->tp_hash != NULL)
    return (*tp->tp_hash)(v);
```

Fig. 9. The execution of the hijacked function pointer
Discussion

- Scalable context-sensitive static points-to analysis
  - Unavailable before mid 2018.
  - SUPA has open-sourced in July, 2018.
  - SUPA is based on SVF. It is both scalable and context-, flow-, and field-sensitive.
  - Though, it fails to solve large number of pointers and returns imprecise points-to result.

- The offline log can be automated to verify and update CFG.
  - We keep it as a future work.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Out of Budget</th>
<th>Empty Points-to Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of ICTs</td>
<td>SUPA</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>54</td>
<td>639</td>
</tr>
<tr>
<td>403.gcc</td>
<td>46</td>
<td>544</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>22</td>
<td>1645</td>
</tr>
<tr>
<td>447.dealll</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>450.soplex</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>37</td>
<td>143</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

TABLE V: Failed cases of SUPA and the improvements of our type-based matching. Column 3, 4, and 6 show the largest EC sizes for SUPA and the type-based matching. SUPA works for all other benchmarks.
Conclusion

● Call-site sensitivity as context-sensitive CFI
● CFI-LB:
  ○ Fast and secure inline reference monitor
  ○ Complete and precise CFG
● Performance overhead:
  ○ 2.7% for the forward-edge protection
  ○ 4.8% for the full protection.
● Open-source: [https://github.com/mustakcsecuet/CFI-LB](https://github.com/mustakcsecuet/CFI-LB)
Q/A

Thank you
Protected Call Stack Solutions

A number of solutions are available and in progress:

- **SafeStack**
  - Separates return addresses and others safe data into a separate safe stack.
  - Published in OSDI’14 and adopted by LLVM in 2015 (still active in service, clang-9.0).

- **ShadowCallStack**
  - Stores only the array of return addresses (contrast to SafeStack).
  - Available for aarch64 in LLVM (from clang-7.0).

- **Intel Control-flow Enforcement Technology (CET)**
  - Would add native support to use a shadow stack to store/check return addresses at call/return time.
  - Would not suffer from race conditions and would not incur the overhead of function instrumentation.
  - Recent update is on May 2019. (active research)
Path Sensitivity Pitfalls

Two most popular system use for runtime path tracing are:

- **Intel Last Branch Record (LBR)**
  - record the most recent 16/32 branches.
  - requires kernel modification and runtime access.
  - applied in PathArmor.
    - limited to paths before seven sensitive syscalls.

- **Intel Processor Tracing (PT)**
  - introduce for offline debug purpose
  - record whole execution path history but in compressed packets
  - requires kernel module for expensive decompression
  - applied in PittyPat, GRIFFIN, etc:
    - entire execution leading to the sensitive syscalls
  - also applied in μCFI:
    - reports data loss problem from Intel PT
    - for gcc, dealII, povray, omnetpp and xalancbmk spec benchmark
A technical pitfalls on SVF/SUPA:

class A {
    public:
        int f();
        int (A::*x)();
};

int A::f() { return 1; }

void ctx() {
    A a;
    a.x = &A::f;
    (a.*(a.x))();
}

##<> Source Loc:
Ptr 4295 PointsTo: {empty}

A conceptual pitfalls on SVF/SUPA:

fnptr instArr[] = {fn_add, fn_mul, fn_mov};
void buildInst(int i){
    instArr[i]();
}
void createAdd(){
    buildInst(0);
}
void createMov(){
    buildInst(2);
}

##<> Source Loc: (no context found)
Ptr 4056 PointsTo: {fn_addr, fn_mul, fn_mov}
**Improvements over Small Set Seeds**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>dyn-CFG-r</th>
<th>dyn-CFG-t</th>
<th>con-CFG-t</th>
<th>dyn-CFG-r \ dyn-CFG-t</th>
<th>dyn-CFG-r \ con-CFG-t</th>
<th>Discovered</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>1374</td>
<td>449</td>
<td>1051</td>
<td>925</td>
<td>323(23.51%)</td>
<td>602</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>12</td>
<td>12</td>
<td>16</td>
<td>0</td>
<td>0(0%)</td>
<td>0</td>
</tr>
<tr>
<td>403.gcc</td>
<td>3831</td>
<td>2196</td>
<td>3929</td>
<td>1635</td>
<td>53(1.38%)</td>
<td>1582</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>1882</td>
<td>1102</td>
<td>1833</td>
<td>780</td>
<td>49(2.60%)</td>
<td>731</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>47</td>
<td>3</td>
<td>58</td>
<td>44</td>
<td>1(2.13%)</td>
<td>43</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>0(0%)</td>
<td>0</td>
</tr>
<tr>
<td>464.h2564ref</td>
<td>262</td>
<td>240</td>
<td>473</td>
<td>220</td>
<td>12(4.58%)</td>
<td>4</td>
</tr>
</tbody>
</table>

**TABLE VI:** Comparing concolic and dynamic CFGs. dyn/con-CFG-t is derived from the small test inputs; dyn/con-CFG-r is derived from the large reference inputs. Our localized concolic execution can discover most of the control transfers in the dyn-CFG-r using only the small inputs.