Modular Control Flow Integrity

Peng XU
peng@sec.in.tum.de

Lehrstuhl für Sicherheit in der Informatik,
I20, TU-München

November 28, 2016
Outline

1. Introduction
2. Problems
3. Solution
4. Results
5. Conclusion
Code Reuses Attacks

* Attacks
- Return Oriented Programming
- Jump Oriented Programming
- Blind ROP
- Return to libC (ret2libc)
- Return to ART runtime (ret2libart)
- vTable Hijacking

* Defenses
- Binary-rewriting-based defenses: CCFIR, VTint, etc.
- Source-code-based defenses: MoCFI, MCFI, VTrust, etc.
Problems

* The loss of separate compilation
* Dynamically loaded libraries can not be instrumented
* Traditional CFI policies require all modules, libraries, to be available at instrumentation time
Solution

- Modules can be instrumented and linked independently
- Supports separate compilation
- Treat Model
## Modules

<table>
<thead>
<tr>
<th>Module1</th>
<th>Module 2</th>
<th>.. libc ..</th>
<th>Module3</th>
<th>..</th>
<th>ModuleN</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1::Data</td>
<td>M2::Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M1::Code</td>
<td>M2::Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M1::Auxiliary</td>
<td>M2::Auxiliary</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M1::CFG</td>
<td>M2::CFG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>newM::CFG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Equivalence Classes

* The set of indirect branch targets.
* After partitioning, each equivalence class is assigned a unique number
  - Equivalence-class Number (ECN)
  - branchECN
  - targetECN
  - fine-grained & coarse-grained
Challenges and solution

* Challenges
  - How to update the policy safely and efficiently at runtime in the presence of multi-threading?
  - How to efficiently generate a new policy with high precision when modules are combined?

* Solutions
  - represents the CFG in separate tables outside of code region
  - augments modules with auxiliary type information and uses the type information for CFG generation
Design

* Requirement: fine-grained CFI
* ID Tables
  - Bary Table
  - Tary Table
* Table Access Transaction
  - Check Transaction (TxCheck)
  - Update Transaction (TxUpdate)
* Module linking
ID Tables

* MCFI partitions indirect branch targets into equivalence classes and labels each with an ECN
* ECNs are pulled out of the code section and stored in a runtime data structure
* These tables are maps from addresses to IDs
  - IDs is a unique identifier associated with an address
  - Bary Table: maps from an indirect branch location to the location’s branch ID
  - Tary Table: maps from an address to the identifier of the equivalence class
* Bary Array and Tary Array
Table Access Transaction

* ID Tables may be accessed concurrently by multiple threads. [Write and read at the same time]
* Wraps table operation into transactions and uses Transactional Memory (TM)
  - Check transaction (TxCheck)
  - Update transaction (TxUpdate)
Table Access Transaction - TxCheck

* Executed before an indirect branch
* Given the address where the indirect branch is located
* Given the address which the indirect branch targets
* The transaction reads the branch ID and target ID from ID tables
* Compares the two IDs
Table Access Transaction - TxUpdate

```
TxCheck  {
    popq  %rcx
    movl  %ecx,  %ecx

Try:
    movl  %gs:ConstBaryIndex, %edi
    movl  %gs:(%rcx), %esi
    cmpl  %edi,  %esi
    jne   Check
    jmpq  *%rcx

Check:
    testb  $1,  %sil
    jz    Halt
    cmpw  %di,  %si
    jne   Try

Halt:
    hlt
}
```
Table Access Transaction - TxUpdate

* Executed during dynamic linking
* Given the new IDs generated from the new CFG after linking a library
* The transaction updates the Bary and Tary tables
Table Access Transaction - TxUpdate

* New module is linked, new CFG is generated
* Bary and Tary tables’ entries should be updated
* getBaryECN(code address) - return branch ECN
* getTaryECN(code address) - return target ECN
Table Access Transaction - TxUpdate

```c
void TxUpdate () {
    acquire(updLock);
    globalVersion = globalVersion + 1;
    updTaryTable();
    sfence;
    updBaryTable();
    release(updLock);
}
void updTaryTable() {
    // allocate a table and init to zero
    allocateAndInit(newTbl);
    for (addr=CodeBase; addr<CodeLimit; addr+=4) {
        ecn=getTaryECN(addr);
        if (ecn >= 0) {
            entry=(addr - CodeBase) / 4;
            newTbl[entry]=0x1; // init reserved bits
            setECNAndVer(newTbl, entry,
                        ecn, globalVersion);
        }
    }
    copyTaryTable(newTbl, TaryTableBase);
    free(newTbl);
}
```
Module Linking

* Auxiliary information is used to link modules
* The more information the module carries, the better precision the generated CFG has
* Type information - Types of function pointers and functions
Results - TxUpdate
Results - TxUpdate

![Bar Chart]

The chart above illustrates the results of various benchmarks on different architectures, specifically focusing on x86-32 and x86-64. The benchmarks include `peribench`, `bzip2`, `gcc`, `mcf`, `sjeng`, `liquipwn`, `h264ref`, `mic`, `ibm`, `sphinx3`, and `GeoMean`. Each benchmark is represented by a bar, with the height indicating the performance or success rate. The chart shows a comparison between the two architectures, highlighting differences in performance or efficiency.
## Results - Precision

<table>
<thead>
<tr>
<th>SPEC 2006</th>
<th>x86-32</th>
<th></th>
<th>x86-64</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IBs</td>
<td>IBTs</td>
<td>EQCs</td>
<td>IBs</td>
</tr>
<tr>
<td>perlbench</td>
<td>2250</td>
<td>15492</td>
<td>930</td>
<td>2081</td>
</tr>
<tr>
<td>bzip2</td>
<td>220</td>
<td>515</td>
<td>110</td>
<td>217</td>
</tr>
<tr>
<td>gcc</td>
<td>5215</td>
<td>48634</td>
<td>2779</td>
<td>4796</td>
</tr>
<tr>
<td>mcf</td>
<td>170</td>
<td>468</td>
<td>119</td>
<td>174</td>
</tr>
<tr>
<td>gobmk</td>
<td>2734</td>
<td>11073</td>
<td>709</td>
<td>2487</td>
</tr>
<tr>
<td>hmer</td>
<td>726</td>
<td>4464</td>
<td>401</td>
<td>715</td>
</tr>
<tr>
<td>sjeng</td>
<td>305</td>
<td>1457</td>
<td>207</td>
<td>337</td>
</tr>
<tr>
<td>libquantum</td>
<td>246</td>
<td>754</td>
<td>161</td>
<td>258</td>
</tr>
<tr>
<td>h264ref</td>
<td>1099</td>
<td>3677</td>
<td>493</td>
<td>1096</td>
</tr>
<tr>
<td>milc</td>
<td>441</td>
<td>2443</td>
<td>312</td>
<td>432</td>
</tr>
<tr>
<td>lbm</td>
<td>161</td>
<td>455</td>
<td>112</td>
<td>161</td>
</tr>
<tr>
<td>sphinx3</td>
<td>585</td>
<td>2963</td>
<td>380</td>
<td>589</td>
</tr>
</tbody>
</table>
Conclusion

* Separate Compilation
* Module Instrumentation
* Bray and Tray Tables
* Table Access Transaction
* Dynamic Linking
Q&A?
Thanks!