ReCFA: Resilient Control-Flow Attestation

Yumei Zhang
Xinzhi Liu
Xidian University

Cong Sun
Xidian University
suncong@xidian.edu.cn

Gang Tan
Pennsylvania State University

Xiao Kan
Xidian University

Dongrui Zeng
Pennsylvania State University

Siqi Ma
The University of Queensland

ACSAC 2021
Remote Attestation

Trust Anchor @ prover + Attestation protocol → Authenticity + Freshness

Control-Flow Attestation (C-FLAT, CCS’16)
- A kind of runtime attestations.
- Precisely attest the execution path of the program running at prover.
- Offline: measure the control-flow paths on CFG and store into measurementDB@verifier
- Online: measure the executed path@prover as evidence, and check for validity of this path in the measurementDB.

Problem: Complex program → Path explosion when generating measurementDB
• Control-flow attestation for complex programs (ScaRR, RAID’19)
  • Mitigate path explosion: measuring checkpoint-separated subpaths.

• Limitations:
  • CFG & measurements generation relies on source code.
  • Measuring checkpoints-separated subpaths causes context missing between subpaths.
  • Coarse-grained path diagnoses. Locate only vulnerable subpath but cannot locate the exact vulnerable control-flow events.
• Ideas of ReCFA
  • No offline measurements generation, only binary CFG generated as policy.
  • No source code requirement: binary rewriting of program@prover

• Difficulties:
  • Prover-side events explosion. Require careful design of condensing function $F(\bullet)$.
  • Acceptable runtime overhead at prover. Technical difficulty on rewriting an efficient binary P.
Design of ReCFA

- Runtime path condensing $F(\bullet)$ is conducted by the instrumented code snippets.
- Not every control-flow event has to be instrumented (for efficiency) —— call site filtering.
- Further compression on the control-flow event sequence before sending report —— greedy compression.
Threat Model and Requirements (Similar to C-FLAT, ScaRR)

- Assumptions
  - DEP & trust anchor deployed on prover.
  - Off-the-shelf attestation protocol (out of our scope)

- Attackers can
  - run the program with arbitrary input.
  - read/write the data section of the program.
  - exploit memory corruptions to hijack control flow.

- The verifier **remotely diagnoses** control-flow path leading to control-flow hijacking.
  (different from local CFI) —— usually higher runtime overhead.
• Phase-1: Filter out the skippable direct calls
• Phase-2: Runtime control-flow events folding
• Phase-3: Greedy compression on control-flow event sequence
Multi-phase Control-Flow Condensing of ReCFA

- **Phase-1:** Filter out the skippable direct calls
  - Potential Monitoring Points (PMPs): all function calls, indirect jumps, and returns
  - Intuition: causality relation between consecutive PMPs —— A node is skippable only when none of its predecessors has more than one successor.
  - Build abstract graph from CFG (PMPs as nodes)
  - Detect skippable PMPs (direct calls), only unskippable PMPs are instrumented.
  - Build a mapping M to hold the relation between predecessor and skippable successor (Let the verifier know the skippable node from predecessor node)
• Phase-2: Runtime control-flow events folding
  • Instrumented binary code snippets take action
  • We design
    • where and what to be instrumented
    • what data structure to be manipulated for the events folding
  • Folding to capture the unskipped control-flow events in loops and recursions
  • Path explosion mainly caused by loops and recursions

```c
N0, N1: for(int i=0; i<n; i++) {
    N2: if(i%2==0) {
        N3: privileged();
        N4: else unprivileged();
        N5: endif
    }
N6: ...
Np: privileged() {...}
Nu: unprivileged() {...}
```

(a) Loop Example

• loop entry ($\ell^e$)
• loop exit ($\ell^x$)
• loop body start ($\ell^s$)
• loop body end ($\ell^d$)
Multi-Phase Control-Flow Condensing of ReCFA

- **@loop entry**: push \( \bot \) onto loop stack to demarcate outer/inner loop
- **@loop body start**: start a new stack frame and push its index onto loop stack
- **@loop body end**
  - compare the top stack frame with the stack frames indexed by the loop stack elements above the top-most \( \bot \)
  - pop the top stack frame and its index when duplicated event path found
- **@loop exit**: pop the content of loop stack above top-most \( \bot \), to fold the outer loop

```
N0, N1: for(int i=0; i<n; i++){
    N2: if(i%2==0){
        N3: privileged();
        N4: else unprivileged();
        N5: endif
    }
    N6: ...
Np: privileged() {...}
Nu: unprivileged() {...}
```

(a) Loop Example

Figure 3: Folding Nested Loops
• Phase-2: Runtime control-flow events folding
  • Use the same data structure as loop stack (i.e. conceptually recursion stack) to deal with recursions
  • Use static analysis to identify the recursion cases causing false positives. Skip folding these cases.
• Phase-3: Greedy compression
  • Irrelevant to program structure. On control-flow events sequence
  • Greedy algorithm with a sliding window
    • Add knot information about repeating times
  • Complexity: O(n*BOUND)
    • n: length of events sequence
    • BOUND: size of sliding window
  • Not optimal:
    • $e1e2e1e2e3e1e2e1e2e3$ compressed to $\langle 2, 2 \rangle e1e2e3\langle 2, 2 \rangle e1e2e3$
    instead of $\langle 2, 5 \rangle e1e2e1e2e3$

---

**Algorithm 1: GreedyCompression(p, BOUND)**

```plaintext
idx ← 0; r ← [];
for pos_w ← 0 to length(p) − 1 do
  n_rep ← 0; sz_w ← 1;
  while sz_w < BOUND do
    poschk ← pos_w + sz_w * (n_rep + 1);
    if poschk + sz_w > length(p) ∧ n_rep = 0 then
      break;
    end
    for j ← 0 to sz_w ∧ poschk + j < length(p) do
      if p[pos_w + j] ≠ p[poschk + j] then
        break;
      end
    end
    if j = sz_w then
      n_rep ← n_rep + 1;
    else if n_rep = 0 then
      sz_w ← sz_w + 1;
    else
      knot(idx, (n_rep + 1, sz_w));
      r[idx..(idx + sz_w)] ← p[pos_w..(pos_w + sz_w)];
      idx ← idx + sz_w;
      pos_w ← pos_w + sz_w * (n_rep + 1);
      n_rep ← 0; sz_w ← 1;
    end
  end
  r[idx] ← p[pos_w];
  idx ← idx + 1;
end
compress(r, idx);
```
Context-Sensitive Remote Enforcement

- Verifier-side shadow stack
- Mapping $F$
  - statically for forward edges. The element of $F$ is in form $cs \rightarrow (ca, tgts)$.
  - $cs$: call site address of a forward edge
  - $ca$: address of the call-after point of the call site
  - $tgts$: the set of valid target addresses of the call
- Security policy: $<M,F>$
- For call edge
  - Retrieve the mapping $M$ to find all the skipped events led by this call edge
- For forward edge and its subsequent skipped events
  - Validate the call/branch target (in $tgts$?)
  - Push the call-after point onto the shadow stack
- For returns
  - check “return target =? top element of shadow stack”
Implementation of ReCFA

- Binary-level CFG
  - Derived with TypeArmor. Neutral to different binary CFG generation approaches
- Security policy \(<M,F>\)
  - M: static analysis with Dyninst
  - F: static analysis with TypeArmor
- Edge encoding
  - Indirect branches and returns: a pair of code addresses
  - Direct call: one code address of the call site
- Intel’s MPK protected user-space data structures (loop stack and path stack)
  - CFA data regions only allowed to be written by instrumented code snippets
  - Insert guards at entry and exit points of code snippet
    - The guard notifies the kernel the type of each snippet and the guarded point
  - Kernel-level pairing the consecutive entry/exit signal of guards with the same snipped type
  - Avoid using indirect branches in the code snippets
• SPEC CPU 2006’s C benchmarks (standard workload “test”)
• Binaries build with GCC v7.5.0 and LLVM v10.0.0

**Effect of call-site filtering**
The ratio of reduction ranges 16.1%~57.2% for GCC binaries and 16.1%~54.5% for LLVM binaries. **The overall reduction is around 40.5%.**

<table>
<thead>
<tr>
<th>Program</th>
<th>#d-call</th>
<th>#d-call</th>
<th>#d-call</th>
<th>#d-call</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>orig</td>
<td>skipped</td>
<td>orig</td>
<td>skipped</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>13,793</td>
<td>4,168</td>
<td>13,799</td>
<td>4,179</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>288</td>
<td>134</td>
<td>271</td>
<td>129</td>
</tr>
<tr>
<td>403.gcc</td>
<td>48,610</td>
<td>21,558</td>
<td>48,416</td>
<td>21,412</td>
</tr>
<tr>
<td>429.mcf</td>
<td>31</td>
<td>5</td>
<td>31</td>
<td>5</td>
</tr>
<tr>
<td>433.milc</td>
<td>929</td>
<td>358</td>
<td>929</td>
<td>358</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>8,898</td>
<td>3,150</td>
<td>8,887</td>
<td>3,143</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>2,141</td>
<td>764</td>
<td>2,141</td>
<td>764</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>739</td>
<td>272</td>
<td>739</td>
<td>272</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>407</td>
<td>233</td>
<td>410</td>
<td>222</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>2,070</td>
<td>735</td>
<td>2,070</td>
<td>744</td>
</tr>
<tr>
<td>470.lbm</td>
<td>33</td>
<td>18</td>
<td>33</td>
<td>18</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>2,064</td>
<td>1,075</td>
<td>2,064</td>
<td>1,075</td>
</tr>
<tr>
<td>Overall reduction</td>
<td>40.6%</td>
<td>40.5%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Effect of control-flow events folding**

Average time overhead of instrumented program is **42.3%**
Overall reduction in the control-flow events is **93.2%**

Average attestation speed (E-speed) is **28.2M/s**
Peak D-speed is **2.53MB/s (GCC) and 2.59MB/s (LLVM)**. Average D-speed is **283.0KB/s**

E-speed: speed of the prover generating raw runtime control-flow events
D-speed: speed of the prover generating data that are sent to the verifier

<table>
<thead>
<tr>
<th>Program</th>
<th>$T_{\text{orig}}$ (s)</th>
<th>$T_{\text{instr}}$ (s)</th>
<th>$T_{\text{pr}}$ (s)</th>
<th>$#e_{\text{total}}$ (x10^3)</th>
<th>$#e_{\text{inl}}$ (x10^3)</th>
<th>$#e_{\text{pr}}$ (x10^3)</th>
<th>$Z_e$ (KB)</th>
<th>$T_{\text{orig}}$ (s)</th>
<th>$T_{\text{instr}}$ (s)</th>
<th>$T_{\text{pr}}$ (s)</th>
<th>$#e_{\text{total}}$ (x10^3)</th>
<th>$#e_{\text{inl}}$ (x10^3)</th>
<th>$#e_{\text{pr}}$ (x10^3)</th>
<th>$Z_e$ (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench</td>
<td>1.3</td>
<td>4.0</td>
<td>0.5</td>
<td>25,311.0</td>
<td>15,471.4</td>
<td>15,442.3</td>
<td>519.4</td>
<td>1.6</td>
<td>4.7</td>
<td>0.1</td>
<td>24,884.0</td>
<td>2,855.6</td>
<td>2,830.6</td>
<td>469.1</td>
</tr>
<tr>
<td>biq2</td>
<td>10.3</td>
<td>12.1</td>
<td>0.1</td>
<td>205,593.1</td>
<td>1,804.5</td>
<td>1,742.9</td>
<td>566.6</td>
<td>11.4</td>
<td>13.2</td>
<td>0.1</td>
<td>205,599.3</td>
<td>1,806.7</td>
<td>1,745.1</td>
<td>566.7</td>
</tr>
<tr>
<td>gcc</td>
<td>1.5</td>
<td>3.5</td>
<td>3.4</td>
<td>187,747.3</td>
<td>99,408.6</td>
<td>97,690.7</td>
<td>17,489.3</td>
<td>1.5</td>
<td>3.3</td>
<td>3.5</td>
<td>185,831.5</td>
<td>100,174.0</td>
<td>98,460.3</td>
<td>17,579.9</td>
</tr>
<tr>
<td>mcf</td>
<td>4.0</td>
<td>6.7</td>
<td>0.3</td>
<td>174,799.9</td>
<td>9,767.0</td>
<td>7,690.7</td>
<td>2,195.7</td>
<td>4.4</td>
<td>7.0</td>
<td>0.3</td>
<td>174,799.9</td>
<td>9,767.1</td>
<td>7,690.7</td>
<td>2,241.1</td>
</tr>
<tr>
<td>milc</td>
<td>12.0</td>
<td>13.7</td>
<td>0.7</td>
<td>311,950.1</td>
<td>15.4</td>
<td>15.4</td>
<td>3.9</td>
<td>16.6</td>
<td>18.0</td>
<td>0.1</td>
<td>313,774.1</td>
<td>15.8</td>
<td>15.8</td>
<td>3.0</td>
</tr>
<tr>
<td>gobmk</td>
<td>5.4</td>
<td>7.5</td>
<td>1.6</td>
<td>60,850.8</td>
<td>50,976.7</td>
<td>50,534.1</td>
<td>7,786.2</td>
<td>5.2</td>
<td>7.4</td>
<td>1.6</td>
<td>60,859.8</td>
<td>50,985.4</td>
<td>50,543.0</td>
<td>7,781.5</td>
</tr>
<tr>
<td>hammer</td>
<td>7.4</td>
<td>8.0</td>
<td>0.0</td>
<td>79,139.7</td>
<td>4.7</td>
<td>4.7</td>
<td>2.7</td>
<td>6.8</td>
<td>8.0</td>
<td>0.0</td>
<td>79,139.7</td>
<td>4.7</td>
<td>4.7</td>
<td>2.7</td>
</tr>
<tr>
<td>sjeng</td>
<td>5.6</td>
<td>N/A</td>
<td>N/A</td>
<td>383,144.6</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>5.5</td>
<td>N/A</td>
<td>N/A</td>
<td>378,466.7</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>libquant</td>
<td>0.1</td>
<td>0.1</td>
<td>0.0</td>
<td>1,018.7</td>
<td>24.6</td>
<td>24.6</td>
<td>2.7</td>
<td>0.1</td>
<td>0.1</td>
<td>0.0</td>
<td>1,279.3</td>
<td>24.7</td>
<td>24.7</td>
<td>2.6</td>
</tr>
<tr>
<td>h264ref</td>
<td>27.9</td>
<td>39.6</td>
<td>1.3</td>
<td>2,059,738.2</td>
<td>40,118.8</td>
<td>40,032.9</td>
<td>2,580.7</td>
<td>29.8</td>
<td>41.6</td>
<td>1.8</td>
<td>2,061,382.9</td>
<td>52,545.2</td>
<td>52,459.3</td>
<td>2,976.7</td>
</tr>
<tr>
<td>ливр</td>
<td>2.8</td>
<td>2.8</td>
<td>0.0</td>
<td>0.12</td>
<td>0.03</td>
<td>0.03</td>
<td>0.2</td>
<td>2.5</td>
<td>2.5</td>
<td>0.0</td>
<td>0.12</td>
<td>0.03</td>
<td>0.03</td>
<td>0.2</td>
</tr>
<tr>
<td>sphinx3</td>
<td>2.1</td>
<td>2.3</td>
<td>0.0</td>
<td>34,596.9</td>
<td>842.4</td>
<td>728.4</td>
<td>166.2</td>
<td>2.0</td>
<td>2.3</td>
<td>0.0</td>
<td>34,750.4</td>
<td>836.1</td>
<td>725.0</td>
<td>167.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Avg.</th>
<th>overhead = 43.7%</th>
<th>reduction = 93.2%</th>
<th>overhead = 41.0%</th>
<th>reduction = 93.2%</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-speed</td>
<td>29.2M/s</td>
<td>D-speed = 291.3KB/s</td>
<td>E-speed</td>
<td>27.2M/s</td>
</tr>
</tbody>
</table>

---

\[a\] Small numbers of $\#e$ to two decimal places.
\[b\] 458. sjeng not taken into account.
Effect of $BOUND$ value tuning

Greedy compression time increases exponentially along with the exponential increase of $BOUND$. The increase in the gain of compression is not exponential. Thus small $BOUND$ is preferred.
### Effectiveness of Context-Sensitive Enforcement at Verifier

The average verification speed is 1.03M/s

Incomparable to the speeds of ScaRR. Different definitions of control-flow events

<table>
<thead>
<tr>
<th>Program</th>
<th>GCC</th>
<th>LLVM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$</td>
<td>\mathcal{M}</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>4,289</td>
<td>15,299</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>134</td>
<td>460</td>
</tr>
<tr>
<td>403.gcc</td>
<td>21,879</td>
<td>53,159</td>
</tr>
<tr>
<td>429.mcf</td>
<td>5</td>
<td>83</td>
</tr>
<tr>
<td>433.milc</td>
<td>372</td>
<td>1,591</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>3,191</td>
<td>9,969</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>789</td>
<td>4,074</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>273</td>
<td>1,247</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>234</td>
<td>554</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>750</td>
<td>3,347</td>
</tr>
<tr>
<td>470.lbm</td>
<td>19</td>
<td>74</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>1,078</td>
<td>2,758</td>
</tr>
</tbody>
</table>

| Avg. vrf. speed | 1.27M/s | 0.87M/s |
Real exploits diagnosed by ReCFA
ReCFA’s verifier detects typical exploits detectable by TypeArmor.
Only instrument on a related part of CFG due to the large size of binary

<table>
<thead>
<tr>
<th>Program</th>
<th>Source</th>
<th>Type</th>
<th>Detected?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffmpeg</td>
<td>CVE-2016-10190</td>
<td>heap corruption</td>
<td>✓</td>
</tr>
<tr>
<td>Apache httpd</td>
<td>PoC exploit of [15]</td>
<td>heap corruption</td>
<td>✓</td>
</tr>
<tr>
<td>Nginx</td>
<td>PoC exploit of [15]</td>
<td>heap corruption</td>
<td>✓</td>
</tr>
</tbody>
</table>

Available: [https://github.com/suncongxd/ReCFA](https://github.com/suncongxd/ReCFA)
THANKS

Thanks for listening