Methodologies for Quantifying (Re-)randomization Security and Timing under JIT-ROP

Salman Ahmed, Ya Xiao
Ph.D. Students
Virginia Tech

Gang Tan
Professor
Penn State University

Kevin Snow
Co-Founder
Zero Point Dynamics

Fabian Monrose
Professor
UNC at Chapel Hill

Daphne Yao
Professor
Virginia Tech
JIT-ROP is a powerful attack technique known for bypassing fine-grained ASLR

- Repeated code pointer leak from a single leak

Does JIT-ROP completely break fine-grained ASLR?

- How much broken the fine-grained ASLR is?
- Are there still good elements of fine-grained ASLR?
Motivation

In-depth questions regarding the impact of fine-grained ASLR on code reuse attacks is not clear

Unclear to choose re-randomization intervals.
Key Questions to Answer

1. What impact do fine-grained ASLR have on the Turing-complete expressiveness of JIT-ROP payloads?

2. How do attack vectors (e.g., code pointer leaks) impact the code reuse attacks?

3. How would one compute the re-randomization interval effectively to defeat JIT-ROP attacks?
Our Measurement Approach

We emulate parts of the JIT-ROP attack.

Fine-grained versions are built using Zipr1, SR2, CCR3, MCR4, Shuffler5

Apps/libs run workloads & intentionally leak code pointers.

Gadgets, havest-time, and libc pointers are stored.

We evaluated 5 fine-grained ASLR tools using 20 applications, and 25 dynamic libraries.

1https://git.zephyr-software.com/opensrc/irdb-cookbook-examples
2https://github.com/immunant/selfrando
3https://github.com/kevinkoo001/CCR
4https://github.com/securesystemslab/multicompiler
5https://github.com/orgs/columbia/teams/shuffler-ro
Why NOT Launching JIT-ROP Exploits?

We did not launch JIT-ROP exploits due to

(1) low scalability,

(2) low reproducibility, and

(3) inaccurate measurement issues

Need **specific, relevant, and measurable** metrics

Require **systemic** measurement methodologies
Our Metrics and Methodologies
We identify **FOUR** security metrics and design **FOUR** measurement methodologies.

### Security metrics

1. **Attack time**
2. **Gadget availability**
3. **Quality of gadget**
4. **Number of libc pointers**

### Methodologies

1. **Upper bound**
   - We determine the upper bound for re-randomization intervals using attack time and gadget availability metrics.

2. **Attack surface reduction**
   - We determine attack surface reduction using availability of gadgets.

3. **Security impact**
   - We quantify the security impact of defenses by varying attack time and attack vectors and using quality of gadget chain.

4. **Critical module**
   - We determine the critical module of a binary using the number of libc pointers.
Our Gadget Availability and Gadget Quality Metrics

We represent each gadget using **TWO footprints**.

1. **Minimum footprint gadgets**: mov rax, rbx; ret;
2. **Extended footprint gadgets**: mov rax, rbx; add rax, rsi; ret;

We compute **gadget corruption rate** based on the register corruption in extended footprint gadgets.

mov edx, dword ptr [rdi];
**mov eax, edx;** → core instruction
shr eax, 0x10;
xor eax, edx;
ret;

We combine **FOUR** sets of gadgets for the gadget availability metric.

1. Turing-complete (TC) gadget set
2. Priority gadget set
3. MOV TC gadget set
4. Payload gadget set

We compute gadget corruption rate based on the register corruption in extended footprint gadgets.
Our Threat Model

- Stack Canary, W⊕X, RELRO
- **Fine-grained ASLR**
- A leaked pointer is available
- No CFI + XoM + CPI
- Attack model: JIT-ROP

Decoupling them helps one better understand the individual factor’s security impact.
Our Findings
Our Finding 1: Computing Re-Randomization Upper Bound

The upper bound* ranges from 1.5 to 3.5 seconds in our tested applications such as nginx, proftpd, firefox, etc.

* May vary with machine configurations

Turing-complete gadget set with a timeline for new gadget type leaks.
Our Finding 2: Quantification of Attack Surface Reduction

Single-round **instruction-level** randomization limits up to 90% gadgets and restricts Turing-complete operations.

<table>
<thead>
<tr>
<th>Randomization schemes</th>
<th>Granularity</th>
<th>↓ (%) MIN-FP</th>
<th>↓ (%) EX-FP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main executables</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst. level rando. [50]</td>
<td>Inst.</td>
<td><strong>79.7</strong></td>
<td><strong>82.5</strong></td>
</tr>
<tr>
<td>Func. level rando. [25]</td>
<td>FB</td>
<td>27.63</td>
<td>36.55</td>
</tr>
<tr>
<td>Func.+Reg. level rando. [53]</td>
<td>FB &amp; Reg.</td>
<td>17.62</td>
<td>42.37</td>
</tr>
<tr>
<td>Block level rand. [59]</td>
<td>BB</td>
<td>19.58</td>
<td>44.64</td>
</tr>
<tr>
<td><strong>Dynamic libraries</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst. level rando. [50]</td>
<td>Inst.</td>
<td><strong>81.3</strong></td>
<td><strong>92.2</strong></td>
</tr>
<tr>
<td>Func. level rando. [25]</td>
<td>FB</td>
<td>46.5</td>
<td>43.8</td>
</tr>
<tr>
<td>Func.+Reg. level rando. [53]</td>
<td>FB &amp; Reg.</td>
<td>44.2</td>
<td>43.9</td>
</tr>
<tr>
<td>Block level rand. [59]</td>
<td>BB</td>
<td>20.98</td>
<td>37.0</td>
</tr>
</tbody>
</table>

Reduction of Turing-complete gadget set with different randomization schemes
Our Finding 3: Impact of the Location of Pointer Leakage

No impact on connectivity

Has an impact on the attack time: dense code pages contain diverse set of gadgets

Impact of starting pointer locations on gadget harvesting time.
Our Finding 4: Critical Module Determining

A Stack has **higher risk** than heap or data-segment

Stacks contain **16 more** libc pointers than heaps or data segments on average.
Key Takeaways

Security metrics and methodologies for large-scale evaluations

Methodology to compute effective re-randomization upper bound

High connectivity in code, enabler for JIT-ROP

Instruction-level randomizations limit Turing-complete operations

All leaked pointers are created equal for gadget availability, but not for the time to leaks gadgets
Acknowledgment

We thank the anonymous reviewers and our shepherd for their valuable comments and suggestions. This work was supported in part by the NSF under grant No. CNS-1838271.

Code available on GitHub
https://github.com/salmanyam/jitrop-native
Thank You