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

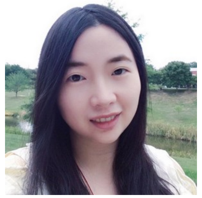
Salman Ahmed, Ya Xiao
Ph.D. Students
Virginia Tech

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 Attack and Fine-grained ASLR

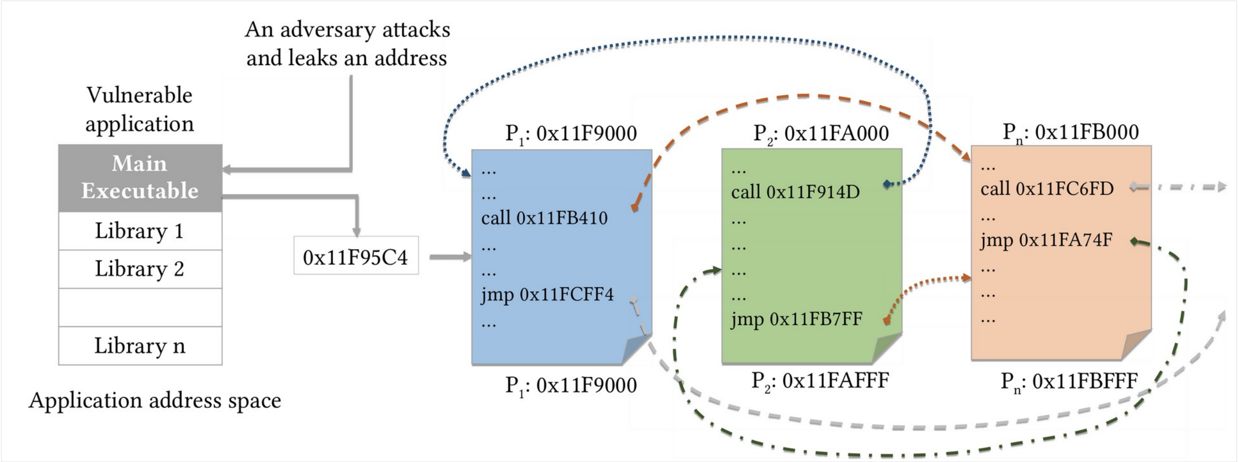
JIT-ROP is a powerful attack technique known for bypassing fine-grained ASLR

- 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?



Just-In-Time Return-Oriented Programming (JIT-ROP)

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?

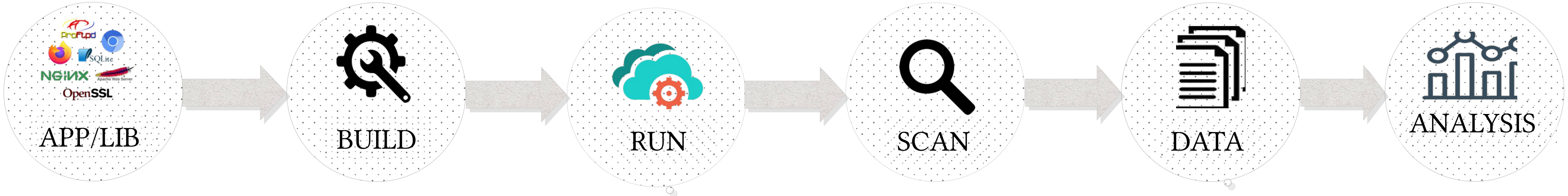
Our Measurement Approach

We emulate parts of the JIT-ROP attack.

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

Fine-grained versions are built using Zipr[1], SR[2], CCR[3], MCR[4], Shuffler[5]

JIT-ROP code harvest is performed



Apps/libs run workloads & intentionally leak code pointers.

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

[1]https://git.zephyr-software.com/opensrc/irdb-cookbook-examples
[2]https://github.com/immunant/selfrando
[3]https://github.com/kevinkoo001/CCR
[4]https://github.com/securesystemslab/multicompiler
[5]https://github.com/orgs/columbia/teams/shuffler-ro

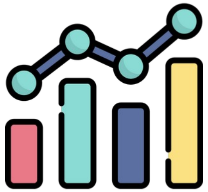
Why NOT Launching JIT-ROP Exploits?

We did not launch **JIT-ROP** exploits due to

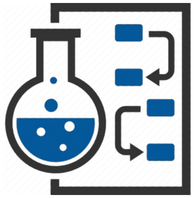
(1) low scalability,

(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 determining 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;
```

Details of gadget sets in the paper

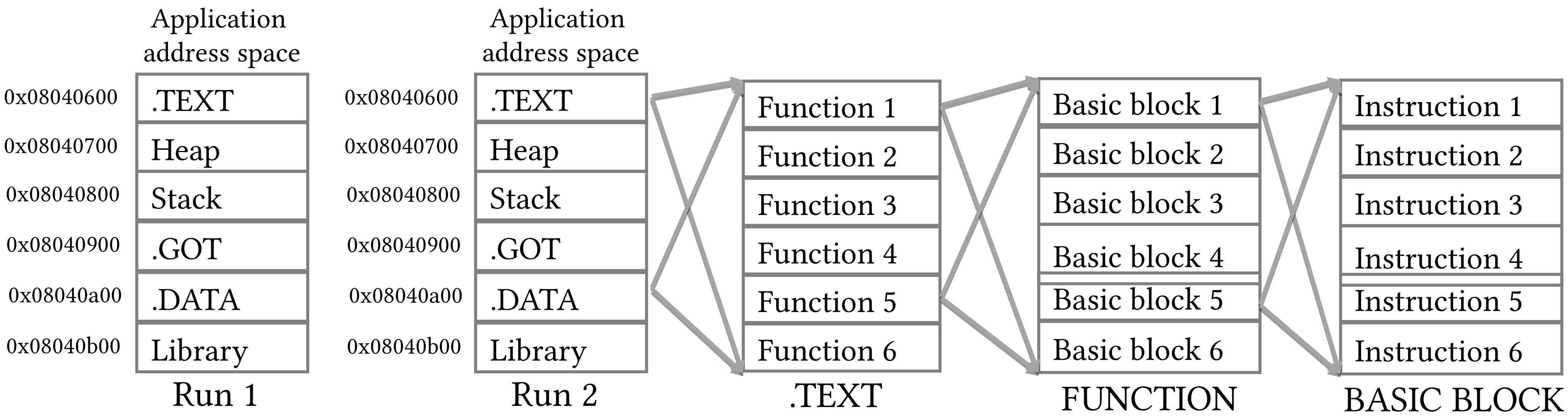
We combine **FOUR** sets of gadgets for the gadget availability metric.

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

Our Threat Model

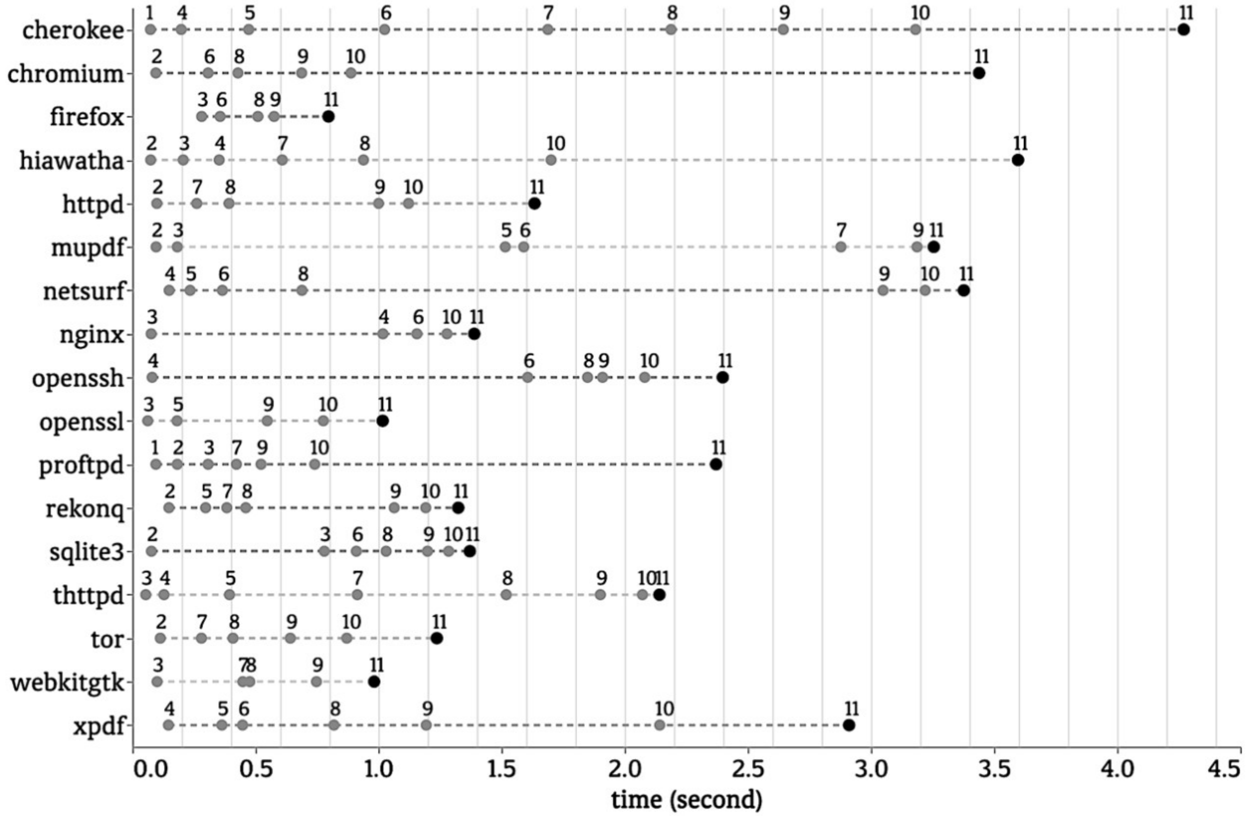
- Stack Canary, W⊕X, RELRO
- **Fine-grained ASLR**
- A leaked pointer is available
- No CFI + XoM + CPI
- Attack model: JIT-ROP

Decoupling them helps one better understand the individual factor's security impact.



Our Findings

Our Finding 1: Computing Re-Randomization Upper Bound



Turing-complete gadget set with a timeline for new gadget type leaks.

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

	Time to leak all gadget types	
Gadget set	**Minimum (s)**	**Average (s)**
TC	2.2	4.3
Priority	1.5	3.5
MOV TC	3.5	5.3
Payload*	2.1	4.8
Average	**2.3s**	**4.5s**

* May vary with machine configurations

Our Finding 2: Quantification of Attack Surface Reduction

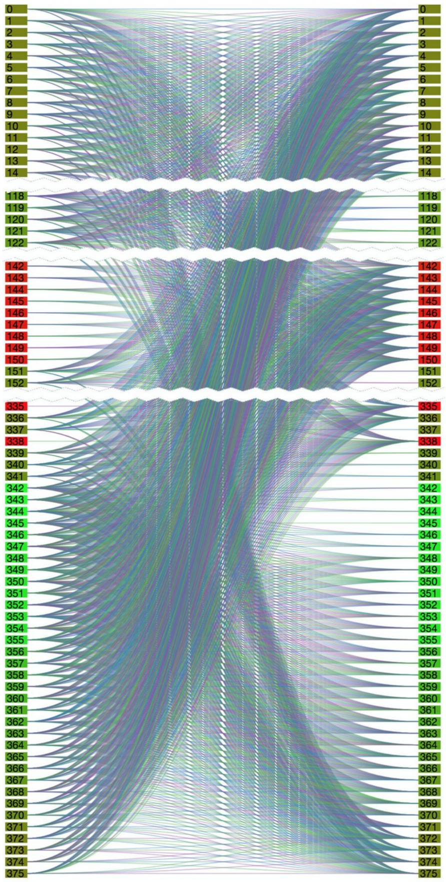
Single-round **instruction-level** randomization limits up to **90%** gadgets and restricts Turing-complete operations.

Randomization schemes	Granularity	$\Downarrow$ (%) MIN-FP	$\Downarrow$ (%) EX-FP
Main executables			
Inst. level rando. [50]	Inst.	**79.7**	**82.5**
Func. level rando. [25]	FB	27.63	36.55
Func.+Reg. level rando. [53]	FB & Reg.	17.62	42.37
Block level rand. [59]	BB	19.58	44.64
Dynamic libraries			
Inst. level rando. [50]	Inst.	**81.3**	**92.2**
Func. level rando. [25]	FB	46.5	43.8
Func.+Reg. level rando. [53]	FB & Reg.	44.2	43.9
Block level rand. [59]	BB	20.98	37.0

Reduction of Turing-complete gadget set with different randomization schemes

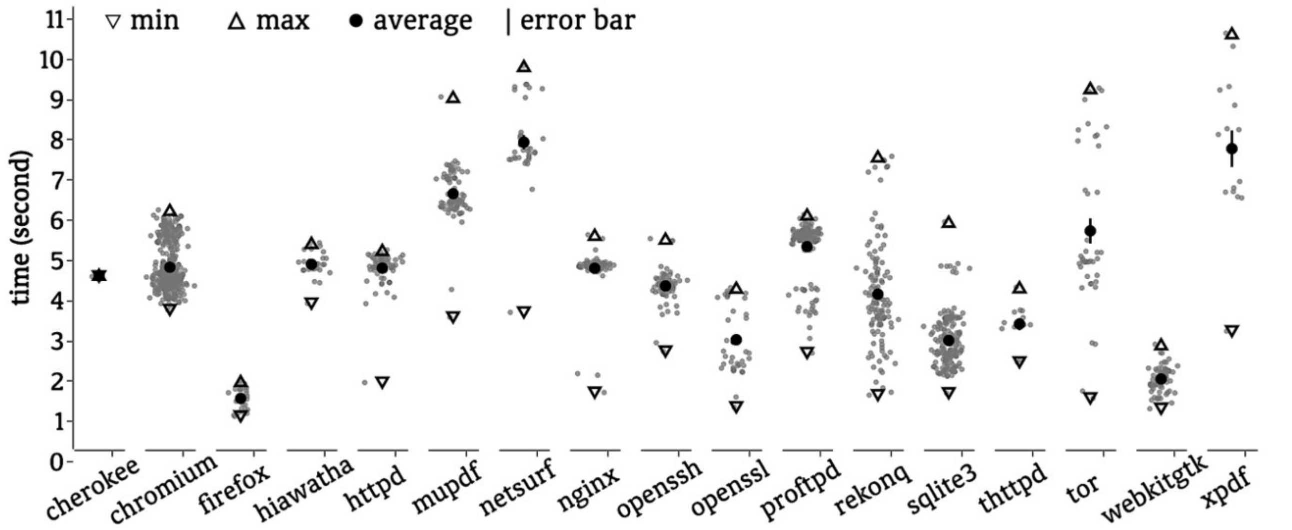
Our Finding 3: Impact of the Location of Pointer Leakage

No impact on connectivity



Connectivity of libc

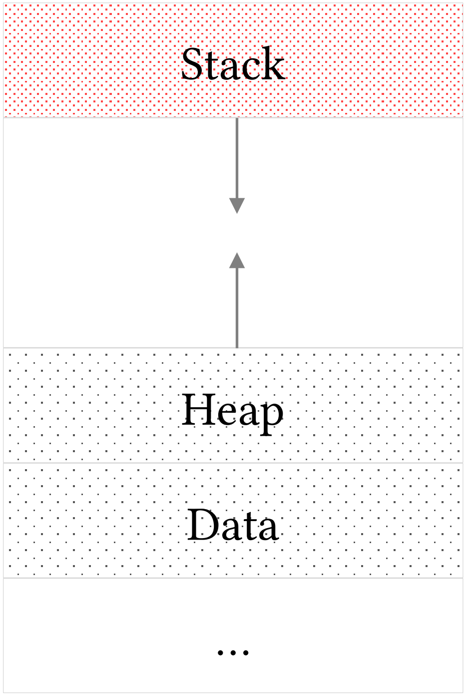
Has an impact on the attack time: dense code pages contain diverse set of gadgets

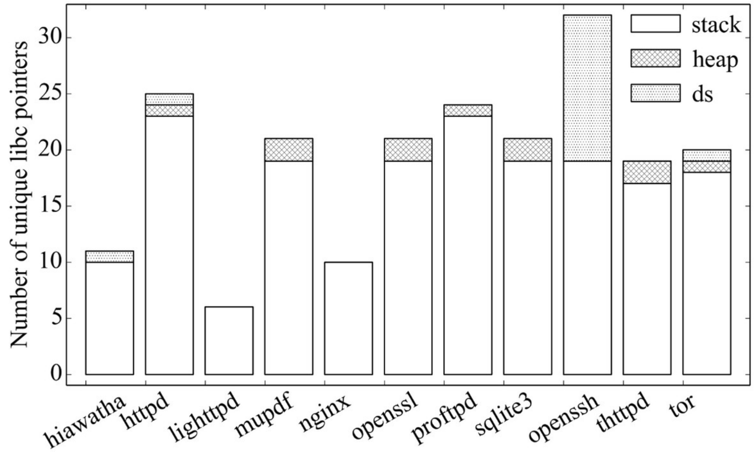


Impact of starting pointer locations on gadget harvesting time.

Our Finding 4: Critical Module Determining

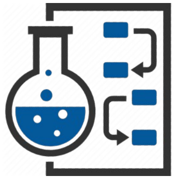
A Stack has higher risk than heap or data-segment





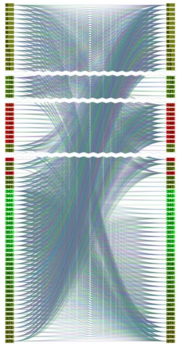
Stacks contain 16 more libc pointers than heaps or data segments on average.

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 availabile on GitHub
https://github.com/salmanyam/jitrop-native

Thank You