Poster: Methodologies for Quantifying (Re-) Randomization Security and Timing under JIT-ROP

Md Salman Ahmed*, Ya Xiao*, Gang Tan†, Kevin Snow‡, Fabian Monrose§, Danfeng (Daphne) Yao*
*Computer Science, Virginia Tech, †Computer Science and Engineering, Penn State University,
‡Zeropoint Dynamics, LLC, §Computer Science, UNC at Chapel Hill
{ahmedms, yax99, danfeng}@vt.edu, gtan@cse.psu.edu, kevin@zeropointdynamics.com, fabian@cs.unc.edu

***Abstract*—Just-in-time return-oriented programming (JIT-ROP) technique allows one to dynamically discover instruction pages and launch code reuse attacks, effectively bypassing most fine-grained address space layout randomization (ASLR) protection. However, in-depth questions regarding the impact of code (re-)randomization on code reuse attacks have not been studied. For example, *how would one compute the re-randomization interval effectively by considering the speed of gadget convergence to defeat JIT-ROP attacks?*; *how do starting pointers in JIT-ROP impact gadget availability, time of gadget convergence, and the Turing-complete (TC) expressive power of JIT-ROP payloads?*; We conduct a comprehensive measurement study and provide methodologies to measure JIT-ROP gadget availability, quality, and their TC expressiveness, as well as to empirically determine the upper bound of re-randomization intervals in re-randomization schemes. Experiments show that the locations of leaked pointers used in JIT-ROP attacks have no impacts on gadget availability, but have an impact on the time for accumulating the TC gadget set. The time ranges from around 0.89 to 5 seconds in our tested applications. Our results also show that instruction-level single-round code randomization thwarts current gadget finding techniques under the JIT-ROP threat model.**

Introduction. JIT-ROP [15] is a powerful attack that enables one to reuse code even under fine-grained ASLR. JIT-ROP attacks can discover new code pages dynamically, by leveraging control-flow transfer instructions, such as `call` and `jmp` and construct exploit payloads at runtime. Re-randomization techniques [16], XoM [12]/XnR [2] style defenses, Code Pointer Integrity (CPI) [11], and Control-Flow Integrity (CFI) [1] have potential to defeat JIT-ROP attacks. However, from a **defense-in-depth** perspective, it is important for a critical system to deploy multiple complementary security defenses in practice due to the potential failure of a single defense. Thus, despite the strong security guarantees of CFI with the latest advancement (e.g., MLTA [13]), our ASLR investigation is still extremely necessary. It is also useful and necessary to isolate various defense factors to better understand the individual factor's security impact. Otherwise, it might be too complicated to interpret the experimental results. This is the reason we chose to focus on ASLR defenses in this work and omit other such as CFI, CPI, and XoM/XnR style defenses.

In this study, we report our experimental findings on various aspects of code (re-)randomization that impact code reuse attacks, e.g., in terms of interval choices, code pointer leakage, gadget availability, gadget convergence, speed of convergence, and gadget chain formation. We use the term *gadget convergence* for a set gadgets to indicate that the set of gadgets has met the criteria of the Turing-complete gadget set. In ROP literature, the Turing-complete gadget set refers to a set of gadgets that cover the Turing-complete operations including memory, assignment, arithmetic, logic, control flow, function call, and system call [14]. Our evaluation involves up to 20 applications including 6 browsers, 1 browser engine, and 25 dynamic libraries.

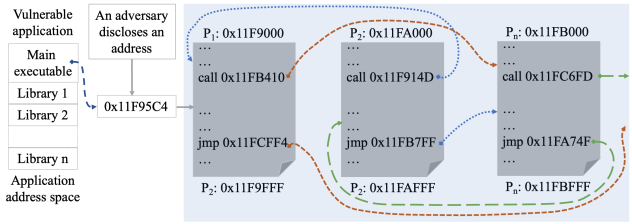

Fig. 1. An illustration of the recursive code harvest process of JIT-ROP [15].

Measurement Methodologies. We designed a measurement mechanism that allows us to perform JIT-ROP's recursive code page discovery (Figure 1) in a scalable fashion. We focus on the native execution of JIT-ROP that allows us to evaluate re-randomization timing and multiple fine-grained ASLR conditions such as the function-level [5], block-level [10], machine register-level [9], [6], and instruction-level [8] code randomization. Since native execution is faster than WebAsm/JavaScript [7], our timing results measured using the native execution is also conservatively applicable for the scripting environments. We manually extracted 21 types of gadgets including the Turing-complete gadget set from various attacks [15], [4], [3]. We measure the occurrences of 15 selected gadgets under fine-grained code randomization schemes. We use ropper[1], an offline gadget finder tool, under coarse-grained ASLR. Under fine-grained ASLR, we write a tool to recreate the native JIT-ROP exploitation process, including code page discovery and gadget mining. We also measure the upper bound[2] for re-randomization intervals by determining how much the code harvest process takes to find the Turing-complete gadget set. To measure the quality of individual gadgets, we perform a register corruption analysis for each gadget. To determine the risk associated with a stack/heap/data segment, we count the vulnerable library pointers in a stack, heap or data segment. We aslo assess the effect of compiler optimizations on the gadget availability.

Evaluation results. We implemented a JIT-ROP native code module. All experiments are performed on a Linux machine

[1] https://github.com/sashs/Ropper

[2] We define that the upper bound of a re-randomization scheme is the maximum amount of time between two consecutive randomization rounds that prevent an attacker from obtaining the Turing-complete gadget set.

with Ubuntu 16.04 LTS 64-bit operating system. We write several Python and bash scripts for automating our analysis and measurement process.

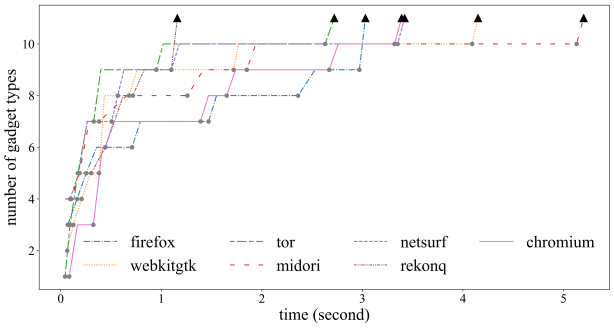

Fig. 2. Gadget convergence with trajectory lines. Each ▲ and ● represent a re-randomization upper bound and re-randomization interval, respectively.

Re-randomization upper bound. Using our methodologies, we measure the upper bounds of re-randomization intervals for 19 applications including 6 browsers and 1 browser engine. Figure 2 shows the upper bound for the browsers along with their trajectory lines of convergence. We observe that the upper bound ranges from around 0.89 to 5 seconds. We call the upper bound as the "best-case" re-randomization interval from a defender's perspective because the defender has to re-randomize by the time of the interval, if not sooner.

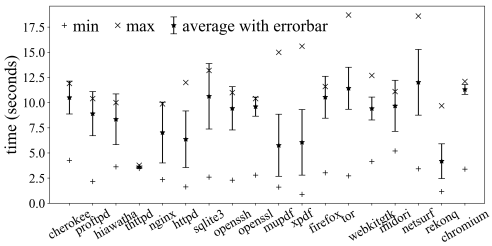

Fig. 3. Min, max, and average time needed to harvest the TC gadget set.

Impact of the Location of Pointer Leakage. We measure the impact of pointer locations on JIT-ROP attack capabilities, by comparing the number of gadgets harvested and the time of harvest under different *starting* pointer locations. For all applications, we observe that the pointer's location does not have any impact on gadget availability. However, the times needed to harvest the TC gadget set vary from one pointer to another. Figure 3 shows the minimum, maximum, and average time required for gadget convergence for different applications and browsers. For some code pointers, the code harvest process takes significantly shorter times than the average times due to the fact that some code pages with diverse set of gadgets are accessed sooner for those code pointers.

Other findings. Function, basic-block, or machine register-level randomization preserves TC expressiveness, however, instruction-level randomization does not. Our findings suggest that current fine-grained randomization solutions do not impose significant gadget corruption. In addition, a stack has a higher risk of revealing dynamic libraries than a heap or data segment due to the higher number of *libc* pointers, on average more than 16 in stack than heaps or data segments.

Conclusion. We presented general methodologies for quantitatively measuring ASLR security under the JIT-ROP threat model and conducted a comprehensive measurement study. One method is for experimentally determining the upper bound of re-randomization intervals. Another method is for computing the number of various gadget types and their quality.

Acknowledgment. This work is supported by ONR Grant N00014-17-1-2498 and DARPA/ONR N66001-17-C-4052.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[2] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, p. 1342.

[3] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 227–242.

[4] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses." in *USENIX Security Symposium*, 2014, pp. 385–399.

[5] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi, "Selfrando: Securing the tor browser against de-anonymization exploits," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 4, pp. 454–469, 2016.

[6] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 763–780.

[7] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *ACM SIGPLAN Notices*, vol. 52, no. 6. ACM, 2017, pp. 185–200.

[8] W. H. Hawkins, J. D. Hiser, M. Co, A. Nguyen-Tuong, and J. W. Davidson, "Zipr: Efficient static binary rewriting for security," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 559–566.

[9] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2013, pp. 1–11.

[10] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-assisted code randomization," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 461–477.

[11] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity." in *OSDI*, vol. 14, 2014, p. 00000.

[12] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *Acm Sigplan Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[13] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1867–1881.

[14] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[15] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 574–588.

[16] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization." in *OSDI*, 2016, pp. 367–382.

Poster: Methodologies for Quantifying (Re-)Randomization Security and Timing under JIT-ROP

Md Salman Ahmed[1], Ya Xiao[1], Gang Tan[2], Kevin Snow[3], Fabian Monrose[4], Danfeng (Daphne) Yao[1]
[1]Computer Science, Virginia Tech, [2]Computer Science & Eng., Penn State, [3]Zeropoint Dynamics, LLC, [4]Computer Science, UNC at Chapel Hill
{ahmedms, yax99, danfeng}@vt.edu, gtan@cse.psu.edu, kevin@zeropointdynamics.com, fabian@cs.unc.edu

1. Motivation

- From **defense-in-depth** perspective, deployment of multiple defenses is necessary.
- Feasibility analysis and **quantitative evaluation** of these defenses are also necessary.
- Despite the strong security of other defenses such CFI [1], CPI [2], XoM [3]/XnR[4] style defenses, **investigations on ASLR** are extremely necessary.
- **General methodologies** for measuring **ASLR security and timing** using various metrics are necessary.
- ASLR security and timing metrics can include the following:
  - interval choices
  - gadget availability
  - speed of convergence
  - code pointer leakage
  - gadget convergence
  - gadget chain formation

2. Challenges

- How to **quantify** the impact of fine-grained ASLR or code randomization or re-randomization schemes.
- How to **quantify** the quality of a gadget chain.

We report our experimental findings on various aspects of code (re-)randomization that impact code reuse attacks: (i) upper bound for re-randomization interval choices, (ii) code pointer locations, (iii) gadget availability, (iv) gadget convergence, (v) speed of convergence, and (vi) gadget chain formation.

3. Approach and Experimental Design

- We identify **21 JIT-ROP gadgets** including the **Turing-complete (TC)** gadget set.
- We measure gadgets and re-randomization timing with (re-)randomization schemes enforced by **5** tools for **20** applications and **25** libraries utilizing a native JIT-ROP implementation. Figure 1 shows JIT-ROP's **recursive code harvest** process.
- To measure the upper bound, we record the time for a JIT-ROP attacker to harvest the TC gadget set.
- To measure the impact of code pointer locations, we run code harvest process starting from different code pointer locations and track gadget convergence and convergence time.
- We measure the impact of single-round randomization by comparing the number of TC gadgets available in randomized and non-randomized versions of an application.
- We measure the gadget quality using register corruption rate.

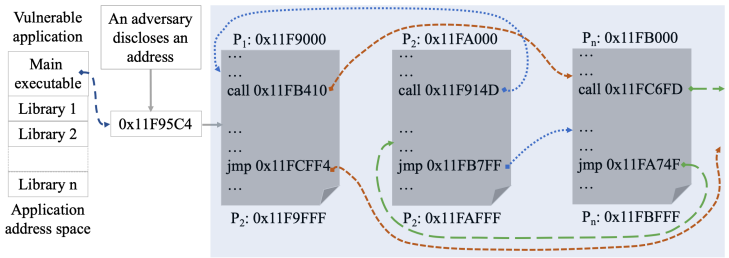

Figure 1: An illustration of the **recursive code harvest process** of JIT-ROP

4. Evaluation Results

- We found that re-randomization upper bound varies from 0.89 to 5 seconds in our test applications and browsers on our machine.

Figure 2 shows the re-randomization upper bounds and intervals along with the trajectory lines.

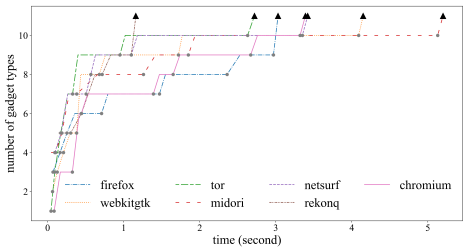

Figure 2: Gadget convergence with trajectory lines. Each ▲ and ● represent a re-randomization upper bound and re-randomization interval, respectively.

- For all applications, we observe that the pointer's location does not have any impact on gadget availability. However, the times needed to harvest the TC gadget set vary from one pointer to another (Figure 3).

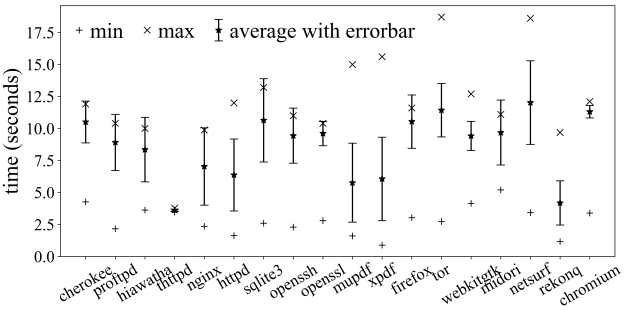

Figure 2: Min, max, and average time needed to harvest the TC gadget set.

- Instruction-level randomization does not preserves TC expressiveness of JIT-ROP payloads. Our findings suggest that current fine-grained randomization solutions do not impose significant gadget corruption. In addition, a stack has a higher risk of revealing dynamic libraries than a heap or data segment due to the higher number of *libc* pointers, on average more than **16** in stack than heaps or data segments.

5. Conclusion

We presented multiple general methodologies for quantitatively measuring the ASLR security under the JIT-ROP threat model and conducted a comprehensive measurement study. One method is for computing the number of various gadget types and their quality. Another method is for experimentally determining the upper bound of re-randomization intervals. The upper bound helps guide re-randomization adopters to make more informed configuration decisions.

[1]. Abadi, Martín, et al. "Control-flow integrity principles, implementations, and applications." ACM Transactions on Information and System Security (TISSEC) 13.1 (2009): 1-40.

[2]. Kuznetzov, Volodymyr, et al. "Code-pointer integrity." The Continuing Arms Race: Code-Reuse Attacks and Defenses. 2018. 81-116.

[3]. Lie, David, et al. "Architectural support for copy and tamper resistant software." Acm Sigplan Notices 35.11 (2000): 168-177.

[4]. Backes, Michael, et al. "You can run but you can't read: Preventing disclosure exploits in executable code." Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. 2014.

[5]. Snow, Kevin Z., et al. "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization." 2013 IEEE Symposium on Security and Privacy. IEEE, 2013..

This work is supported by ONR Grant N00014-17-1-2498 and DARPA/ONR N66001-17-C-4052.

VIRGINIA TECH

Yao Group on Cyber Security
http://yaogroup.cs.vt.edu/

NDSS SYMPOSIUM 2020