Shadowclone: Blocking DOP Attacks with Compile time Stack Layout Randomization

Yunjie Pan
University of Michigan
Ann Arbor, MI, USA
panyj@umich.edu

Shibo Chen
University of Michigan
Ann Arbor, MI, USA
chshibo@umich.edu

Cheng Chi
University of Michigan
Ann Arbor, MI, USA
chicheng@umich.edu

Yifan Guan
University of Michigan
Ann Arbor, MI, USA
yfguan@umich.edu

***Abstract*—Control-flow hijacking attacks have been hard to deploy due to the widespread adoption of control-flow attack defenses such as Control-flow Integrity (CFI). This fact has led to a wide deployment of exploiting non-control data, which are not protected by CFI defenses. Non-control data attacks can be used to corrupt critical data or leak sensitive information. Furthermore, data-oriented programming (DOP) is able to achieve Turing-complete computation capabilities without leaving the control-flow graph of the programs. In this paper, we present a compile time stack layout randomization scheme- Shadowclone -to thwart and detect DOP attacks effectively. Shadowclone generates randomized clones of vulnerable target functions and randomly selects one copy of clones to execute during runtime. In addition, we also insert compile-time random canaries into stack variables and check its integrity before the function returns. In the evaluation section, we show that our approach can thwart and detect DOP attacks efficiently by limiting attacker's success chance to less than 1%. Shadowclone also has low performance overhead when the program is small and has few function calls.**

I. Introduction

Most of high performance applications are written in C or C++. Those languages are inherently prone to memory corruption. Memory corruption is a common approach to deploy control-flow attacks where the attackers can execute arbitrary code on the victim machine. Many approaches have been proposed to protect programs against control-flow attacks, such as Control Flow Integrity (CFI [1]) and Code Pointer Integrity (CPI). These approaches help confine programs to the program-specified control flow graph.

However, non-control data attacks can reuse existing control-flow to manipulate data without violating constraints introduced by defenses against control-flow attacks. One special non-control data attack: Data-Oriented Programming (DOP) [2] enables attackers to execute instructions within legitimate control-flow graph by corrupting non-control data repetitively. DOP has been proved to be Turing-complete, and can construct expressive non-control data exploits for arbitrary programs.

Most DOP attacks make the most of the deterministic nature of stack layout of the program to grant attackers the ability to generate attack payloads. Smokestack [3] proposed an effective stack layout randomization technique to thwart DOP attacks. However, it does not detect attacks while they are happening.

In our project, we present Shadowclone, a compile-time stack-layout randomization technique that can both thwart and detect DOP attacks.

II. ShadowClone Runtime stack layout randomization

Randomizing the code layout and the address space can mitigate control-flow attacks as they require the absolute address of the assets, either code or sensitive data. Even though CFI defenses are shown to be effective at mitigating control-flow attacks, DOP attacks are not stopped by control-flow denfenses. DOP attacks use the relative distance between local variables to make use of memory vulnerabilities. Thus, DOP attacks could circumvent static stack layout randomization and random padding schemes.

A. Design Objectives

The main objective of our project is to provide a practical approach to thwart DOP attacks effectively. We aim to only induce little to none performance overhead. To meet this goal, our solution should follow the requirements listed below:

- Provide a compile time stack randomization scheme to mitigate DOP attacks.
- Use canary to detect attacks.
- Have low performance on both CPU-bound and I/O-bound applications.
- Be compatible with legacy code.

B. Threat Model

In this paper, we assume that the attack initially has no idea of the configuration we generate for stack allocations. By exhaustive runs of its target program, attackers could eventually figure out what all M different configuration are. Since our project focuses on DOP attacks, we assume that control-flow defenses, like Control flow integrity (CFI) have been deployed so that programs are not vulnerable to control-flow attacks. In all, the threat model of Shadowclone is as follows:

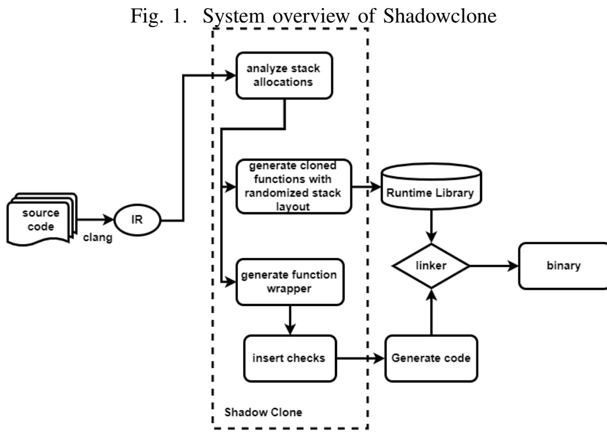
- In most cases attackers would need at least two functions to deliver attacks so that the attackers have enough DOP gadgets. However, as shown in the evaluation section, Shadowclone is effective even if the attacker only needs to compromise one function.

- Attackers could use the semantics of the underlying program to reverse engineer a randomized stack layout of a function based on a disclosed data that allows the adversary to instantiate a runtime attack on future calls of the same function.
- The attacker can perform an infinite number of attempts before being detected by the system so that he could figure out all random stack layouts given enough time.

C. Overview of Shadowclone

Non-control data attacks tamper with leaked sensitive memory, which is not directly used in control flow. In Data-oriented programming (DOP) [2], it has been proved that non-control data attacks can be Turing-complete. If the relative address of stack allocations in known to the attack, a memory corruption vulnerability is exploited to corrupt non-control data. The successive steps will utilize memory vulnerability to control key variables to execute payload instructions. Therefore, our model needs to ensure that the absolute address as well as the relative distance of the stack allocations are unknown to the attacker. Our Shadowclone achieves this by randomly calling functions with different stack layouts for each function invocation. The overview of Shadowclone is shown in 1. To avoid performance overhead when randomizing stack layout at runtime, different functions with randomized and unique stack layout is generated at compile time.

Fig. 1. System overview of Shadowclone



D. Analyzing and Randomizing Stack Allocation

In the analysis phase, we identify and collect all stack allocations in every function of the program. There is no stack allocation alignment requirement. Because the randomization of stack allocations is to reorder `AllocaInst` randomly. Figure 2 shows the instrumental introduced by Shadowclone.

a) Generating Random Permutation: Given N `AllocaInst`, there are $N!$ possible permutations of the allocation instructions. In this stage, we randomly choose M ($M < N!$) permutations for each function in the program. To do this, we randomize the order of `AllocaInst` many times until there are M unique configurations. The M configurations are represented as M cloned functions with unique and randomized stack layout. If there are only one or two stack allocations in a function, there is no need to do the randomization.

b) Transforming Function Wrapper: In this step, we transform the original source code to a function wrapper which calls cloned functions with different stack layout configurations generated in the previous step. After deleting all BBs from the original code, we insert Intel `RDRAND` instruction to generate true random number to decide which function we call from the function wrapper. We create callees for each randomized function, setting up arguments to pass in and return value. The randomized calling functions is represented as conditional branches to BBs where each BB is a caller to randomized function.

c) Insert Canary Check: Unlike Smokestack [3] which is lack of non control-flow attack detection, in this step we insert compile-time random canary in stack variables in order to prevent an attacker from knowing canary value. The random canary is generated at function initialization, and stored in a local variable inserted randomly into the stack variables. When the buffer overflows, the random canary will possibly be corrupted, and a failed verification of the canary value will therefore alert of an overflow, which can then be detected and handled. If a DOP attack is detected by canary, the program will terminates and throws an exception.

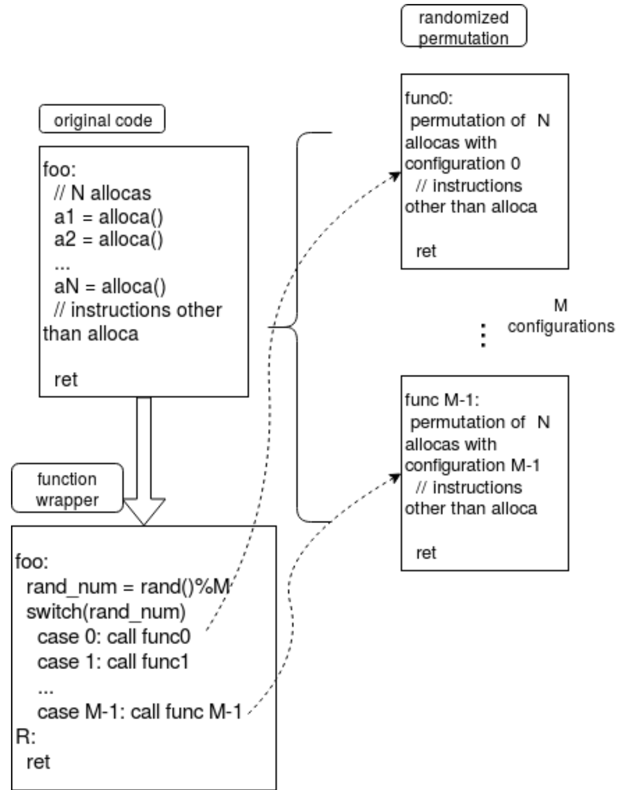


Fig. 2. The overview of the stack layout randomization steps involved in Shadowclone function calls and returns

III. Implementation

We implement our Shadowclone on top of the LLVM 10 compilation framework. Our analysis and instrumental pass are implemented on the LLVM intermediate representation (IR) to randomize stack. We use clang as frontend to generate LLVM IR from source code.

A. Analysis Pass

In analysis pass, we scan the source code to gather all stack allocations of the function which has on-stack memory object. Since all stack allocations are placed in the first basic block(BB) of the function, we only need to go through the first BB.

Then, based on the information of stack allocation we collected, a module pass will clone the function several times. The number of clones has a upper bound as the factorial of the number of stack allocations or a threshold set by the programmer, whichever is smaller. The final analysis pass will randomize the layout of the stack for each cloned instance. So each function has a unique and randomized stack allocation. Notably, no allocation alignment is required here.

B. Instrumental Pass

The instrumental pass will transform the original function to a function wrapper. The first step is to remove all BBs from the original function. Then we insert a BB to generate a random number to determine which randomized function to call. The third step is to create callee instructions of the cloned and randomized functions. Then the control flow is reconstructed, generating control basic blocks to jump to the callee function based on the random number generated at runtime. The final instrumentation pass inserts canary checks before every return location to detect attacks that will overwrite stack buffers.

IV. Evaluation

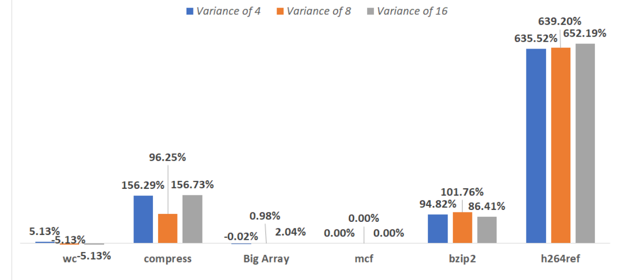
This section presents the detailed performance and security evaluation of Shadowclone. We run our experiments on an Intel Xeon 6126 processor running Ubuntu 18.04 Linux with 256 GB of memory. We conduct both performance evaluation and security analysis in order to find out the Pareto optimal trade-off between performance and security.

A. Performance Evaluation

For performance evaluation, we evaluate Shadowclone over 6 different benchmarks. Three of them are in-House built benchmarks(*big_array*, *wc*, and *compress*) and the other three are from Spec06 benchmark suite (*bzip2*, *h264ref*, and *mcf*) [4].

We compile each benchmark with three different configurations: each function has up to 4, 8, and 16 clones respectively. Since our binary generation and execution involves randomness, for each benchmark and each configuration, we would generate three different binaries, and for each binary we would execute it for 3 times. The performance of a specific benchmark with a specific configuration is the average execution time of the total 9 runs (3 binaries, each with 3 runs).

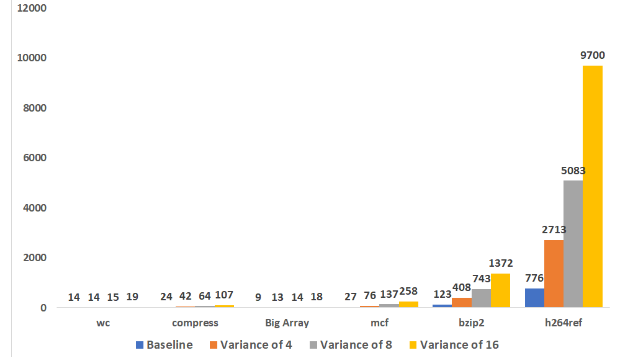
Fig. 3. Performance Overhead of Shadowclone over 6 Benchmarks



Initially, we would expect the execution time of Shadowclone to be significantly faster than that of Smokestack [3] because we are doing more during compile time and less during the runtime. As shown in fig.3, while in 3 out of 6 benchmarks we evaluate, we see little to none performance overhead, Shadowclone induces significant performance overhead in the rest three benchmarks. For benchmark *h264ref* which has a large code size with a great number of function calls, the performance overhead can go up to 6x.

With in-depth analysis, we find out that Shadowclone has significant impact on spatial and temporal locality of the I-Cache and speculative execution - features that modern processors heavily rely on to improve performance.

Fig. 4. Code Size Overhead of Shadowclone over 6 Benchmarks in KB



Spatial Locality: As shown in fig.4, the code size of the generated binary increases proportionally to the number of variances we generate for each function. Due to the significant increase in the code size and the fact that programs generated by Shadowclone would randomly execute an arbitrary region, the spatial locality is severely impacted. Fig.5 illustrates that the number of I-Cache misses increases along with the increasing code size.

Temporal locality and speculative execution: At the beginning of each function call, the program would first generate a random value and use it to branch into one of the basic blocks which contains a clone of the original function. Due to the nature of randomness, it is theoretically impossible for the processor to predict which clone would be executed

Fig. 5. Percentage of # of I-Cache Misses Increase of Shadowclone over 6 Benchmarks

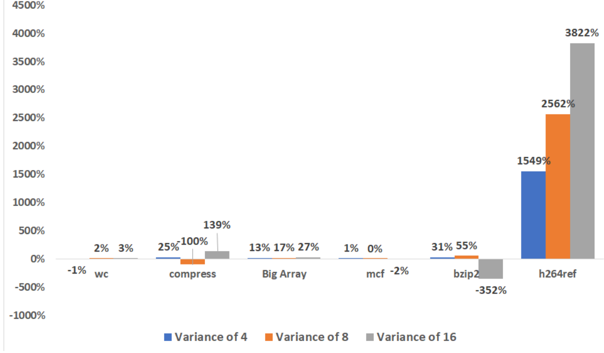
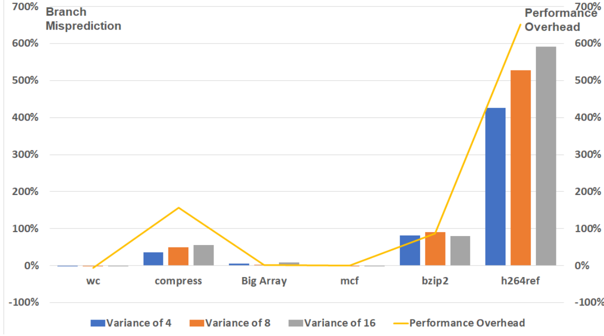


Fig. 6. Percentage of # of Branch Misprediction Increase of Shadowclone over 6 Benchmarks



and thus makes the execution sequential in that code region. Since we randomly choose one clone to run among the total *M* clones, the chance that a single clone would be executed in two consecutive runs is small, for which we also lose the temporal locality of the functions. As shown in fig.6, the number of branch misprediction has a strong correlation to the performance overhead, which indicates that the impact of Shadowclone on speculative execution and temporal locality contributes significantly to the performance overhead.

The insights we gain from the analysis above show that the performance of small programs with fewer function calls would not be severely influenced by Shadowclone whilst that of the large programs with a great number of function calls would suffer much. This also leaves a door for future optimizations.

B. Security Analysis

In this section, we evaluate how efficient Shadowclone is at thwarting and also detecting DOP attacks. The metrics we use is the probability that an attacker can successfully compromise 1, 2, or 3 functions by DOP attacks without being detected. To be conservative, we assume the attacker is able to learn the stack layout of a specific clone after only observing it once. We also assume that the attacker is sophisticated enough so that he would never step onto control flow data or other content-sensitive data which would lead to a *sigfault*. Both assumptions we make here are very hard to achieve in real-world environment.

Since both the stack variables and the insertion point of the canary are random, there is always $\frac{1}{2}$ chance that the canary would lay in between the target variable and the vulnerable buffer. Therefore, the attacker would have $\frac{1}{2}$ chance of getting detected for each DOP attack he conducts. Since we assume the attack knows nothing about the stack layout before conducting attack, the probability that the attacker would succeed in the first run and not being detected is $\frac{1}{N!} \cdot \frac{1}{2}$, where N is the total number of stack variables in a single function. There are on average 10 stack variables in a single function in the spec06 benchmarks we analyzed. After the first run, the attacker would learn at least one configuration existing in the program, therefore the chance that the attacker would succeed without being detected is $\frac{1}{M!} \cdot \frac{1}{2}$, where M is the number of variances existing in our system. We add up the probability of the attacker succeeding in the ith trail and not being detected in the first $i-1$ trails together to get the probability of an attacker would succeed without being detected.

Fig. 7. The Chance of an Attacker Can Compromise 1, 2, 3 Functions without Being Detected

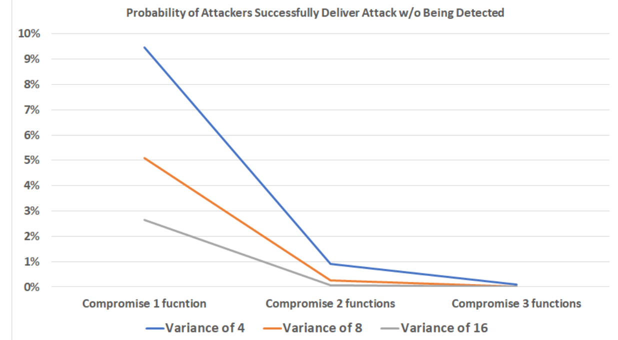


Fig.7 shows that if the attacker only needs to compromise one function, the success chance is less than 3% with 16 variances per function. Furthermore, if the attack needs to compromise more than one function to successfully achieve the malicious goal, the success chance is less than 1% even with only 4 variances.

V. Conclusion

Due to widely successful control-flow attack defenses, non-control data attacks becomes an popular source of attacks against programs. Many existing randomization techniques in the code section of a program cannot effectively thwart DOP attacks. Shadowclone excels at thwarting DOP attacks by randomizing stack frames for functions during compile time. Also, we insert canary to detect attacks, resulting in attackers only have a negligible probability of attacking programs successfully.

Evaluated on several benchmarks, our implementation of Shadowclone in LLVM framework can efficiently thwarts and detects DOP attacks with reasonable slowdown in program execution.

Our proof-of-concept implementation and results demonstrate that Shadowclone can achieve minimal performance

overhead for small programs. However, its performance deteriorates as the size of program gets larger and the program gets more function calls.

Acknowledgment

We are grateful for the guidance from Prof. Scott Mahlke, and assistance from Armand Behroozi, Sunghyun Park. We would also like to offer our special thanks to Prof. Todd Austin for providing experiment equipment and insightful suggestions.

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*, ser. CCS ’05. New York, NY, USA: ACM, 2005, pp. 340–353. [Online]. Available: http://doi.acm.org/10.1145/1102120.1102165

[2] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, “Data-oriented programming: On the expressiveness of non-control data attacks,” in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.

[3] M. T. Aga and T. Austin, “Smokestack: Thwarting dop attacks with runtime stack layout randomization,” in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. Piscataway, NJ, USA: IEEE Press, 2019, pp. 26–36. [Online]. Available: http://dl.acm.org/citation.cfm?id=3314872.3314879

[4] C. D. Spradling, “Spec cpu2006 benchmark tools,” *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 130–134, 2007.