Strengthening Software Diversity Through Targeted Diversification

Vipin Singh Sehrawat[1]
VipinSingh.Sehrawat@utdallas.edu

Yvo Desmedt[1,2]
Yvo.Desmedt@utdallas.edu

[1]Department of Computer Science,
The University of Texas at Dallas, Richardson, USA

[2]Department of Computer Science,
University College London, London, UK

***Abstract*—Code reuse attacks use snippets of code (called gadgets) from the target program. Software diversity aims to thwart code reuse attacks by increasing the uncertainty regarding the target program. The current practice is to quantify the security impact of software diversity algorithms via the number/percentage of the surviving gadgets. Recent attacks prove that only reducing the number of surviving gadgets does not add any security against code reuse attacks. We propose the use of the count/percentage of usable and surviving gadgets as the metric to quantify the security impact of software diversity algorithms. We present a novel software diversity algorithm, named *NOP4Gadgets*, that leaves 0.012% and 14.35% surviving and usable gadgets, respectively. NOP4Gadgets performs targeted diversification, concentrated around the potential Return Oriented Programming (ROP) gadgets. The performance overhead of NOP4Gadgets is 1% for the SPEC CPU2006 benchmark suite.**

Keywords–Software diversity, Return Oriented Programming, Code reuse attack, Targeted diversification.

I. INTRODUCTION

Code reuse attacks use snippets of code (called gadgets) from the target program and libraries. They allow the attacker to bypass the modern defence mechanisms like Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR). ROP [1] and Jump Oriented Programming (JOP) [2] are two forms of code reuse attack. Since the introduction of code reuse attack with return-to-libc [3], numerous defense mechanisms and tools have been proposed for its detection and/or prevention [4]–[12].

Code reuse attacks are facilitated by "Software Monoculture", that is the practice of running the same software on a large number of machines. So, if an attack is successful against the software then it can be used to compromise all machines which run that software. Software monoculture and the over-reliance on certain pieces of software, whether they are operating systems or applications, have been cited [13], [14] to increase the likelihood and severity of widespread security compromises.

Cohen [15] proposed *program evolution* as a solution to software monoculture. Program evolution implies that a program should evolve into different but semantically similar versions of itself. The primary goal of software diversity techniques is to achieve efficient (with minimum overhead) program evolution. Larsen et al. [16] provide a detailed survey of the known software diversity techniques. The common approach to quantify the security impact of software diversity techniques is by counting the number of surviving gadgets, which are the functionally similar gadgets present at the same positions within the diversified copies of an executable. Recent attack by Snow et al. [17] shows that only reducing the number of surviving gadgets does not add any security against code reuse attacks. In this paper, we propose a new metric to quantify the security impact of software diversity techniques. Our approach is to use the count/percentage of both, usable (intended and unintended ROP gadgets that can be used in an attack) and surviving gadgets to better quantify the security impact. Section III explains our metric and its advantages in detail.

In this paper, we present a novel software diversity algorithm, called NOP4Gadgets, that performs targeted diversification. Unlike existing no-op insertion implementations [18]–[21] NOP4Gadgets decides the type(s) of no-op(s) and the probability of no-op insertion based on the current and previous machine instructions, written by the compiler. By virtue of targeted diversification, NOP4Gadgets successfully reduces the number of both surviving and usable gadgets. Note that there are known techniques like G-Free [6] and Return Less Kernels [7] that target usable gadgets, but those are not software diversity techniques as they are not geared towards producing large numbers of diverse versions of the given executable. Existing no-op insertion based software diversity techniques do not perform targeted diversification, instead they all rely only on randomized no-op insertion. Also, none of the known implementations provides any analysis about its security impact in terms of the usable gadgets' statistics.

NOP4Gadgets leaves less than 0.80% surviving gadgets, and incurs a negligible overhead of 1% for the SPEC CPU2006 benchmark suite. Unlike existing no-op insertion implementations NOP4Gadgets focuses only on the potential ROP gadgets, thus avoiding unnecessary work. Goktas et al. [22] showed that gadgets with more than 30 instructions are also usable. So, while measuring the security impact of NOP4Gadgets we fixed the maximum gadget length to 200 bytes. We also developed a stronger version of NOP4Gadgets, which on average leaves

 ISBN: 978-1-61208-493-0

0.012% surviving gadgets and 14.35% usable gadgets, and incurs a negligible additional overhead of 0.651%. A drawback of software diversity is that it offers multiple attack surfaces. This provides the attacker with the option to attack the more vulnerable version(s). NOP4Gadgets leaves similar number of surviving and usable gadgets in each diverse version of an executable. Hence, no version is 'weaker' than the other.

The rest of the paper is organized as follows. Section II gives a background on code reuse attacks and no-op insertion. In Section III, we present and discuss our metric for quantifying the security impact of software diversity algorithms. In Section IV, we present our software diversity algorithm, named NOP4Gadgets, that performs targeted diversification, concentrated around the potential ROP gadgets. Section V gives the detailed security impact and performance overhead analysis of NOP4Gadgets. Section VI and Section VII give the future work and conclusion, respectively.

II. BACKGROUND

A. Code Reuse Attack

Code reuse attacks use snippets of code (called gadgets) from the target executable/library. Return-to-libc [3] is the early form of code reuse attack in which the attacker reuses entire functions of `libc`. Code reuse attacks allow the attacker to defeat DEP by avoiding direct code injection. ROP and JOP are the two classes of code reuse attack. The gadgets used in ROP and JOP end with return and jump instructions, respectively.

Checkoway et al. [23] demonstrated the effectiveness of code reuse attack by successfully compromising a Direct Recording Electronic (DRE) voting machine by using ROP gadgets. An ROP attack uses buffer overflow to overwrite the stack with a series of return addresses that point to known ROP gadgets. By carefully positioning data on the stack the attacker can execute the gadgets in any desired order. A sequence of ROP gadgets that are executed in a predetermined order is called ROP chain. Table I shows an example ROP chain that adds the contents of two memory addresses and stores the result at a third memory address.

TABLE I. ROP CHAIN EXAMPLE

Address	ROP Gadget
0x00401077	pop eax pop ebx ret
0x00400795	mov eax, [eax] ret
0x00400ef6	mov ebx, [ebx] ret
0x0040136c	add eax, ebx pop ecx ret
0x004011ad	mov [ecx], eax ret

The general steps of an ROP attack are:

1) Analyze the code of the target executable and related libraries for aligned or unaligned instruction sequences, that end with a return instruction.
2) Filter the discovered ROP gadgets according to the desired attack.
3) Use buffer overflow to inject the starting addresses of the gadgets, as well as the addresses of any required data onto the stack.
4) Overwrite the return address with the address of the first gadget of the ROP chain.

Once the return instruction of the first gadget gets executed, the value stored in the instruction pointer, `eip`, is updated to the address of the first gadget. After the initial gadget is executed, its return instruction updates the value of `eip` to the address of the second gadget in the chain. In this manner, each gadget returns control to the next gadget in the chain. Automated tools like "Return-Oriented toolkits" [24] can be used to construct arbitrary attack codes using ROP gadgets.

JOP is more subtle than ROP, a jump instruction only performs an unidirectional control flow transfer. To manipulate the control flow the attacker uses a special gadget called the *dispatcher gadget*. To initiate the attack, a buffer overflow is used to jump to the dispatcher gadget. After executing, each gadget returns the control back to the dispatcher gadget which forces a jump to the next gadget in the JOP chain.

Code reuse gadgets can be divided into two broad classes, intended and unintended. Intended gadgets are the ones that consist of proper, aligned instructions, generated by the compiler. On the other hand, the instructions in unintended gadgets start somewhere within the proper instructions. On x86, the number of unintended gadgets always exceeds the number of intended gadgets. But it is harder to use the unintended gadgets because they may include infrequently used instructions and complicated addressing modes. Therefore, only the unintended gadgets of short lengths are considered usable.

B. No-op Insertion

No-ops are short code sequences that when executed have no effect on the registers or the memory. The processor fetches and executes these instructions without any change in the program state. Compilers insert no-ops in the object code to fulfill alignment constraints, and to add timing delays to code fragments [25]. No-op insertion has also been used as a software diversity technique with the aim to reduce the number of surviving gadgets [18]–[21].

No-op insertion can also break existing ROP gadgets, especially the unintended ones. The x86 instruction set is very irregular and the lengths and formats of the instructions depend on the first byte (opcode). Even a single byte inserted inside the byte array of a gadget can significantly alter it. Our algorithm, NOP4Gadgets, uses this property of no-op insertion and targets all potential ROP gadgets. NOP4Gadgets performs targeted diversification, the bulk of which is done within the potential ROP gadgets. Therefore, even if the attacker jumps into the binary at an arbitrary point and executes some unaligned instructions, before reaching the return instruction she encounters the no-ops inserted by NOP4Gadgets. Hence, the execution is forced to realign with the actual code. Figure 1 shows how inserting a no-op instruction at the right position can break an existing ROP gadget.

 ISBN: 978-1-61208-493-0

ROP gadget before no-op insertion:

03 3B	89 E1	89 C3
ADD edi, [ebx]	MOV ecx, esp	MOV ebx, eax

Gadget removed after no-op insertion:

03 3B	89 E1	90	89 C3
ADD edi, [ebx]	MOV ecx, esp	NOP	MOV ebx, eax

Figure 1. Removing ROP gadget by using no-op insertion

1) Why was the gadget removed: Inserting no-op before the return instruction changes the length and decoding of the instruction. For example, the no-op(s) inserted before a return opcode 'C3' may cause it to be decoded as an operand of the preceding instruction. This removes the gadget as now the sequence of instructions does not end with the return instruction. This gadget removing effect is more profound with the x86 instruction set because of the higher number of unintended gadgets.

III. Our approach to quantify the security impact

All known software diversity techniques quantify their security impact via the number/percentage of surviving gadgets present in the diverse versions. The purpose of software diversity is to produce diverse versions of the given program. The diversity achieved by any technique can be measured by recording the disparities in the diverse versions. Thus, counting the number of surviving gadgets is the correct method of quantifying the diversity. But when it comes to measuring the security impact against specific class(es) of attack(s), it is pivotal that we take into account the nature and various components of the attack. Snow et al. [17] showed that concentrating only on reducing the number of surviving gadgets does not add any security against ROP attacks. Therefore, using only the number of surviving gadgets is not the right method to quantify the security impact against code reuse attacks.

We propose the count/percentage of usable and surviving gadgets as the metric to quantify the security impact of software diversity algorithms. The following list gives the various advantages of our approach over the current approach.

1) Sophisticated attacks [17] do not make any assumptions about the surviving gadgets, but all attacks need some minimum number of usable gadgets. Therefore, the number of remaining usable gadgets must be a prime criterion in measuring the security impact against code reuse attacks.
2) We know that surviving gadgets are a good measure of diversity, but the count of the usable gadgets gives the exact number of gadgets available to the attacker.

IV. Our Algorithm (NOP4Gadgets)

We present a novel software diversity algorithm, named NOP4Gadgets, that performs targeted diversification, focused on the potential ROP gadgets. Unlike existing no-op insertion implementations [18]–[21] NOP4Gadgets decides the type and number of no-op(s), along with the probability of no-op insertion, based on the current and previous machine instructions, written by the compiler. Any set of harmless (which do not create new gadgets) no-op instructions can be used to implement NOP4Gadgets. But as we compare the performance overhead of our algorithm with "Profile-guided NOP insertion" (PNOP) [18], we used the same set of no-ops as used in the PNOP implementation. Table II lists the no-op instructions used in the implementation.

```
- I: MachineBasicBlock Iterator
- BB: MachineFunction Iterator
- MI: MachineInstr pointer to I
- NopTable: Table of candidate no-ops (Table II)
- bool pre1 ← false, pre2 ← false
1: procedure NOP4Gadgets
2:     if (MI->getOpcode() == ret) then
3:         call PrecedingInstNOP(BB, I)
4:         call CandidateInstNOP(BB, I)
5:     else
6:         call RandomInstNOP(BB, I)
7: procedure PrecedingInstNOP(BB, I)
8:     if (I > BB->begin()) then
9:         I ← I-1
10:        pre1 ← true
11:        if (I > BB->begin()) then
12:            I ← I-1
13:            pre2 ← true
14:    call PrecedingNOPs(pre1, pre2, BB, I)
15: procedure PrecedingNOPs(pre1, pre 2, BB, I)
16:    if pre2 == true then
17:        With probability p2 call insertRandom(BB, I)
18:        I ← I+1
19:    if pre1 == true then
20:        With probability p1 call insertSpecific(BB, I)
21:        I ← I+1
22: procedure CandidateInstNOP(BB, I)
23:    with probability q1 call insertSpecific(BB, I) once
                              OR
24:    with probability q2 call insertSpecific(BB, I) twice
                              OR
25:    with probability q3 call insertSpecific(BB, I) thrice
26:    I ← I+1
27: procedure RandomInstNOP(BB, I)
28:    with probability p call insertRandom(BB, I)
29: procedure insertSpecific(BB, I)
30:    insert a random 2 byte no-op instruction
31: procedure insertRandom(BB, I)
32:    insert a random no-op instruction
```

Figure 2. NOP4Gadgets as an LLVM pass

We implemented NOP4Gadgets as a *MachineFunctionPass* of Low Level Virtual Machine (LLVM 3.5). Our backend pass identifies the return instructions as they are being written by the compiler and invokes the appropriate no-op insertion function(s) accordingly. Figure 2 gives the pseudocode of NOP4Gadgets. NOP4Gadgets examines the two instructions that immediately precede the return instruction. The user can customize this process to examine any number of preceding instructions. This feature of NOP4Gadgets can be used to identify the type of the potential ROP gadget and then configure the

 ISBN: 978-1-61208-493-0

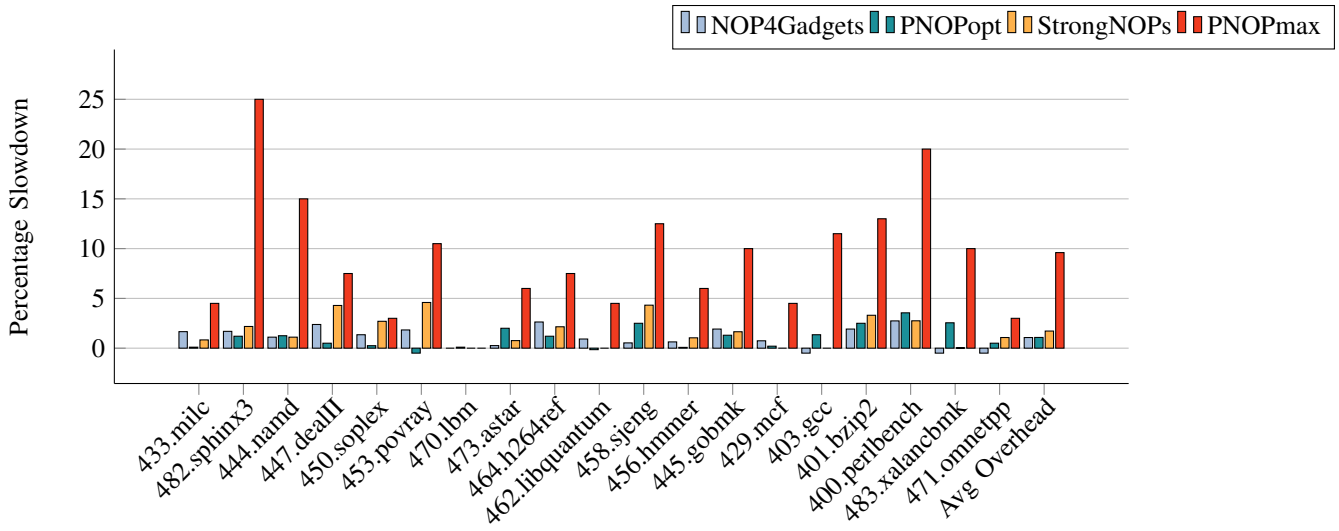

Figure 3. SPEC CPU2006 runtime overhead comparison of NOP4Gadgets with Profile-Guided NOP insertion (PNOP)

PNOPopt: **Minimum overhead version of PNOP** | ***PNOPmax:*** **Maximum diversity/security version of PNOP**

behaviour accordingly. For example, if the user wants to target only the LOAD (gadgets that loads value to register) type ROP gadgets then he can configure the algorithm to perform no-op insertion only within the LOAD type gadgets, and ignore the others. NOP4Gadgets uses two functions, *insertSpecific*, which inserts a random two byte no-op and *insertRandom*, which inserts a no-op instruction, randomly selected from Table II. These features provide flexibility and control over the kind and level of security, and enables the user to optimize the performance overhead.

TABLE II. CANDIDATE NO-OP INSTRUCTIONS

Instruction	Opcode
mov esp, esp	89 E4
mov ebp, ebp	89 ED
lea esi, [esi]	8D 36
lea edi, [edi]	8D 3F
no-op	90

Our test results showed that for removing/breaking the existing ROP gadgets, 2 byte long no-ops are more effective than 1 byte no-ops. Thus, if the current instruction is a return instruction `ret`, then NOP4Gadgets inserts random 2 byte no-op(s) before it and the preceding instruction. The number of 2 byte no-ops inserted before `ret` can be one, two or three, and is governed by the probabilities q_1, q_2 and q_3, respectively. The instruction (`pre`) preceding `ret` gets one 2 byte no-op inserted before it with probability p_1. A randomly selected no-op is inserted with probability p_2 before the instruction that precedes `pre`. A randomly chosen no-op is inserted with probability p before the rest of the instructions. The (security impact)/(performance overhead) ratio can be altered by adjusting the various probabilities. For NOP4Gadgets, the probabilities governing no-op insertion are set as: $q_1 = .85$, $q_2 = .05$, $q_3 = 0$, $p_1 = .05$, $p_2 = .05$, $p = .04$.

TABLE III. PERCENTAGE OF SURVIVING GADGETS IN THE SPEC CPU2006 BINARIES BUILT BY NOP4Gadgets COMPILER

Benchmark	Original binary	Surviving %
483.xalancbmk	522681	0.31%
401.bzip2	1425	2.38%
403.gcc	90056	0.58%
429.mcf	1421	4.22%
433.milc	11729	0.86%
444.namd	10487	0.68%
445.gobmk	38927	0.30%
447.dealII	37432	0.89%
450.soplex	28612	0.37%
473.astar	4340	3.68%
482.sphinx3	6509	1.46%
464.h264ref	33763	0.25%
470.lbm	816	6.7%
400.perlbench	39699	0.27%
471.omnetpp	27918	0.49%
456.hmmer	17881	4.25%
458.sjeng	28612	1.29%
462.libquantum	3096	2.90%
453.povray	57208	1.08%
Average	-	0.78%

V. EVALUATION OF NOP4GADGETS

We also developed a stronger version of NOP4Gadgets, named *StrongNOPs*. The various no-op insertion probabilities for StrongNOPs are: $p = 0.05$, $p_1 = 0.5$, $q_1 = 0.10$, $q_2 = 0.55$, $q_3 = 0.35$. This section gives a detailed analysis of the performance overhead and security impact of NOP4Gadgets and StrongNOPs.

 ISBN: 978-1-61208-493-0

TABLE V. USABLE AND SURVIVING GADGETS PRESENT IN THE BINARIES BUILT BY StrongNOPs COMPILER

Program	Usable Gadgets	Surviving Gadgets	Size Increase
Advancemame-1.2	12.96%	0.011%	7.8%
Inkscape-0.48.5	34.68%	0%	2.1%
Scummvm-1.7.0	7.915%	0.033%	5.1%
Ghostscript-9.09	27.56%	0%	6.21%
Wesnoth-1.12.1	17.49%	0%	3.2%

Average Usable Gadgets	Average Surviving Gadgets	Average Size Increase
14.35%	0.012%	4.81%

A. Performance Evaluation

We used SPEC CPU2006 benchmark suite to compute the performance overhead of NOP4Gadgets and StrongNOPs. The average performance overhead of NOP4Gadgets is 1.069%, which is similar to the performance overhead of the minimum overhead version of PNOP [18]. Average performance overhead of StrongNOPs is 1.72%, which much smaller than the 9.5% performance overhead of the maximum diversity/security version of PNOP. Figure 3 shows a comparison of PNOPopt (minimum overhead version of PNOP), NOP4Gadgets, PNOPmax (maximum security/diversity version of PNOP) and StrongNOPs in terms of the percentage slowdown for the SPEC CPU2006 benchmarks.

B. Security impact

We quantified the security impact of NOP4Gadgets and StrongNOPs by using our new metric, that is by using the count/percentage of both usable and surviving gadgets. Goktas et al. [22] showed that ROP gadgets with more than 30 instructions are also usable. So, we set the maximum gadget length to 200 bytes. To count the number of surviving gadgets we wrote a program called *Discoverer*, that uses *ROPgadget* [26] to discover ROP gadgets within the `.text` section of the given executable. It then removes all the no-op instructions from the discovered gadgets and searches for identical gadgets present at the same location within different binaries.

We built 20 copies of the SPEC CPU2006 benchmarks using the NOP4Gadgets compiler. For each benchmark we took the two copies that share the maximum number of surviving gadgets between them. The average percentage of surviving gadgets found in the SPEC CPU2006 binaries was 0.78%. Table III shows the percentage surviving gadgets for each benchmark.

We built five popular open source programs using our StrongNOPs compiler. Table V gives the statistics about the surviving and usable gadgets found in the diversified versions of the programs. Note that in three out of the five programs StrongNOPs left no surviving gadgets. On average, StrongNOPs left only 0.012% surviving gadgets and removed over 85% of the usable ROP gadgets.

VI. Future Work

Current software diversity mechanisms primarily focus only on reducing the number of surviving gadgets. With NOP4Gadgets, we presented a novel approach of combining software diversity with gadget removal. Concentrating on removing/breaking the gadgets naturally reduces the number of surviving gadgets. Table VI lists the five broad categories of ROP gadgets. We plan to extend NOP4Gadgets or devise a similar software diversity technique that removes all gadgets of some specific type(s).

TABLE VI. TYPES OF GADGETS

Gadget type	Semantic	Example
ADJUST	adjust reg./mem.	add eax, 2
CALL	call a function	call [esi]
LOAD	load value to reg.	mov eax, [ebx]
STORE	store to mem.	mov [eax], ebx
SYSCALL	systemcall	sysenter

VII. Conclusion

The current approach to quantify the security impact of software diversity algorithms relies only on the number/percentage of the surviving gadgets. Recent attack by Snow et al. [17] shows that only reducing the number of surviving gadgets does not add any security against code reuse attacks. Hence, the current approach of measuring the security impact is flawed. In this paper, we proposed the use of the count/percentage of usable and surviving gadgets as the metric to quantify the security impact of software diversity algorithms. We argued that the proposed metric has several advantages over the current practice. We also presented a novel software diversity algorithm, named *NOP4Gadgets*, that performs targeted diversification, concentrated around the potential ROP gadgets.

NOP4Gadgets performs the bulk of the diversification within the potential ROP gadgets. It allows the user to target specific class(es) of ROP gadgets, and ignore the others. NOP4Gadgets uses different no-op insertion functions, that are configured to use specific type(s) of no-op instructions. NOP4Gadgets leaves less than 0.80% surviving gadgets, and incurs 1% performance overhead for the SPEC CPU2006 benchmark suite. The stronger version of NOP4Gadgets,

 ISBN: 978-1-61208-493-0

named *StrongNOPs*, breaks more than 85% of the usable ROP gadgets, and incurs a negligible additional performance overhead of 0.651%. On average, StrongNOPs leaves only 0.012% surviving gadgets and 14.35% usable gadgets. We also presented a detailed comparison of NOP4Gadgets with the existing no-op insertion implementations [18]–[21]. Software diversity algorithms that follow our approach of focusing on both usable and surviving gadgets can prove to be a powerful tool against code reuse attacks, especially when combined with other defense mechanisms like G-Free [6] and Control Flow Integrity [8].

REFERENCES

[1] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in Proceedings of the 14th ACM conference on Computer and communications security, 2007, pp. 552–561.

[2] T. Bletsch, "Code-Reuse Attacks: New Frontiers and Defenses," Ph.D. dissertation, North Carolina State University, 2011.

[3] S. Designer, "Return-to-libc attack," Bugtraq, 1997.

[4] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "DROP: Detecting Return-Oriented Programming Malicious Code," in 5th International Conference on Information Systems Security, 2009, pp. 163–177.

[5] L. Davi, A. R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks," in ACM workshop on Scalable trusted computing, 2009, pp. 49–54.

[6] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries," in ACSAC, 2010, pp. 49–58.

[7] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating Return-Oriented Rootkits with Return-less Kernels," in EuroSys, 2010, pp. 195–208.

[8] M. Abadi, M. Budiu, U. Erilingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," in ACM Transactions on Information and System Security (TISSEC), Volume 13, Issue 1, October 2009.

[9] M. Prasad and T. Chiueh, "A Binary Rewriting Defense against Stack-based Buffer Overflow Attacks," in USENIX Annual Technical Conference, 2003, pp. 211–224.

[10] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical Control Flow Integrity and Randomization for Binary Executables," in Proceedings of IEEE Symposium on Security and Privacy, 2013, pp. 559–573.

[11] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, Huijie, and R. Deng, "ROPecker: A Generic and Practical Approach For Defending Against ROP Attack," in 21st Annual Network and Distributed System Security Symposium, 2014.

[12] I. Fratric. (2012, September) ROPGuard: Runtime Prevention of Return-Oriented Programming Attacks. [retrieved: May, 2016]. [Online]. Available: http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf

[13] D. E. Geer, "Monopoly considered harmful," in IEEE Security & Privacy, 2003, pp. 14–17.

[14] M. Stamp, "Risks of monoculture," Communications of the ACM - Homeland security, 2004, vol. 47, p. 120.

[15] F. Cohen, "Operating system protection through program evolution," Computers and Security, 1993, vol. 12, pp. 565–584.

[16] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in SP '14 Proceedings of the 2014 IEEE Symposium on Security and Privacy, 2014, pp. 276–291.

[17] 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," IEEE Symposium on Security and Privacy, pp. 574–588, 2013.

[18] A. Homescue, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided Automated Software Diversity," in Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization, 2013, pp. 1–11.

[19] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, Moving Target Defense. Springer New York, 2011, vol. 54, ch. Compiler-Generated Software Diversity, pp. 77–98.

[20] T. Jackson, "On the Design, Implications, and Effects of Implementing Software Diversity for Security," Ph.D. dissertation, University of California Irvine, 2012.

[21] T. Jackson, A. Homescu, S. Crane, P. Larsen, S. Brunthaler, and M. Franz, Moving Target Defense II. Springer New York, 2013, vol. 100, ch. Diversifying the Software Stack Using Randomized NOP Insertion, pp. 151–173.

[22] E. Goktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in USENIX, 2014.

[23] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham, "Can DREs provide long-lasting security? the case of return-oriented programming and the AVC advantage," in Electronic voting technology/workshop on trustworthy elections, USENIX, 2009.

[24] R. Hundt, E. Raman, M. Thuresson, and N. V. Mao, "An extensible micro-architectural optimizer," in Proceedings of the 9th IEEE/ACM International Symposium on Code Generation and Optimization, CGO, 2011, pp. 1–10.

[25] L. Tang, J. Mars, and M. L. Soffa, "Compiling for niceness:mitigating contention for QoS in warehouse scale computers," in Proceedings of the 10th IEEE/ACM International Symposium on Code Generation and Optimization, 2012, pp. 1–12.

[26] J. Salwan. (2012) ROPgadget - Gadgets finder and auto-roper. [retrieved: May, 2016]. [Online]. Available: http://shell-storm.org/project/ROPgadget/

APPENDIX A
LLVM BACKEND PASS IMPLEMENTATION

We implemented our algorithm as an LLVM MachineFunctionPass. MachineFunctionPass is part of the LLVM code generator that executes on the machine dependent representation of each LLVM function in the program. The next step was to write two no-op insertion functions and add them to LLVM. In the beginning of the MachinFunctionPass we inspect the current machine instruction and depending on whether it is a return instruction or not, we call different function(s). In order to correctly identify the machine instructions the target architecture(s) must be fixed. As NOP4Gadgets is most effective with x86 instruction set, we set x86 as the architecture.

Once it is verified that the current machine instruction is a return instruction, we proceed to the next step that is to move back by one or two instructions, if possible. This is done by comparing the current value of "MachineBasicBlock iterator" with the "MachineFunction iterator" and if possible the MachineFunctionPass moves one or two steps back to the previous instruction(s). Finally, we call one or both no-op insertion functions (insertSpecific and insertRandom).

A. X86::Return Instructions

Below is the list of *LLVM's x86 return instructions*, used in the implementation of NOP4Gadgets.

```
RETQ, IRET64, IRET32, IRET16, LRETQ, RETW,
RETL, RETIL, RETIQ, RETIW, EH_RETURN,
EH_RETURN64, LRETIL, LRETIQ, LRETIW,
LRETL, LRETQ, LRETW
```