Adaptive Just-In-Time Code Diversification

Abhinav Jangda
IIT (BHU) Varanasi
Varanasi, UP 221005
India
abhinav.student.apm11@iitbhu.ac.in

Mohit Mishra
IIT (BHU) Varanasi
Varanasi, UP 221005
India
mohit.mishra.cse11@iitbhu.ac.in

Bjorn De Sutter
Ghent University
Sint-Pietersnieuwstraat 41
9000 Gent, Belgium
bjorn.desutter@elis.ugent.be

ABSTRACT

We present a method to regenerate diversified code dynamically in a Java bytecode JIT compiler, and to update the diversification frequently during the execution of the program. This way, we can significantly reduce the time frame in which attackers can let a program leak useful address space information and subsequently use the leaked information in memory exploits. A proof of concept implementation is evaluated, showing that even though code is recompiled frequently, we can achieved smaller overheads than the previous state of the art, which generated diversity only once during the whole execution of a program.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—code generation, optimization;K.6.5[**Management of Computing and Information Systems**]: Security and Protection—unauthorized access

General Terms

Algorithms, Performance, Experimentation, Security.

Keywords

Recompilation, profiles, NOP insertion.

1. INTRODUCTION

The widespread software monoculture is a major facilitator of cyber attacks. Adversaries can target many users with a single exploit because they all run the same binaries. One type of code reuse attacks, return-oriented programming (ROP) attacks, are particularly hard to defeat [1]. Instead of injecting new code into a binary, ROP attacks exploit a software vulnerability to execute a chain of reusable code snippets called gadgets, thus obtaining execution of any wanted functionality. In response, software diversity has been proposed [2,12,15]. By randomizing a binary's code layout, a memory vulnerability is moved to an a priori unknown location in the binary, thereby bringing down the probability of return-to-libc and return-oriented attacks [1, 2, 6, 21].

Many of the current defense practices are passive and static in nature. Despite techniques such as Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) [3], the cat and mouse games between the attackers and defenders therefore still go on. In many recent attacks, an attacker first lets the program leak information about the code locations in a process' memory space, in order to engineer an adapted attack on the fly.

MTD'15, October 12 2015, Denver, CO, USA © 2015 ACM.
ISBN 978-1-4503-3823-3/15/10…$15.00
DOI: http://dx.doi.org/10.1145/2808475.2808487

Just-In-Time (JIT) compilers generate code during the execution of the program. This is, e.g., the case for Java, which is first precompiled to bytecode and then later compiled JIT to machine code. Similarly, LLVM can first compile code to bitcode, which is then later compiled JIT to native code.

The predictability of JIT compilers makes their security questionable. In practice, minimal variation is introduced into their code generation, e.g., by noise due to sampling-based profiling. So every time the same high-level language code is presented to the compiler, it emits the same native code. The attackers exploit this to their advantage. As JIT compilation fundamentally uses executable memory of which the content is dependent on the input of the users, i.e., the program to be JIT compiled as well as its data, it poses a big security threat.

In this paper, we propose to harden the security of a Java bytecode JIT compiler by regenerating diversified code dynamically. This way, we can significantly reduce the time frame in which attackers can let a program leak useful address space information and subsequently use the leaked information. There are two critical timestamps for an attacker in a typical attack model:

- Time of leakage of useful address space information,
- Time of use of the leaked information in an exploit.

If we can randomize either of the timestamps by making use of diversity, the attacker will likely fail to build an exploit. We consider two aspects of the randomization:

- Security: We adaptively insert NOPs to create diversity.
- Performance: Profile information helps in reducing the overhead while pushing the adaptive diversification.

Using frequent live diversification at random timestamps, we intend to randomize the time frame between the above two timestamps, as well as regenerate a different diversified variant of the binary at these timestamps. With a proof-of-concept implementation in JikesRVM, we show that while securing the JIT compiler further, we also incur smaller overheads as compared to static diversification and static re-diversification by making good use of profile information of the program.

While software diversity helps in protecting "diversified" binaries while "one" is compromised, the idea here is to "protect the one" too that is risking to be compromised.

The paper is structured as follows: Section 2 provides background on code-reuse attacks, Java JIT spraying and memory corruption, and software diversity. Section 3 describes the design of the framework and the proposed solution. Section 4 provides an evaluation and presents results of the conducted. Related work on JIT diversification is discussed in Section 5. Finally, Section 6 draws conclusions and discusses some future work.

2. BACKGROUND

Code reuse attacks allow attackers to execute arbitrary code on a compromised machine. In this, the attacker directs the control flow through existing code without injecting new executable code.

Examples include ROP attacks and return-into-libc attacks. ROP attacks, introduced by Shacham [1], allow attackers to execute arbitrary code by overflowing the stack with a sequence of return addresses pointing to specific parts of the code, called gadgets, in the vulnerable program. A gadget is a valid code sequence that ends with a return instruction. This allows Turing-complete behavior in the target program without injecting and circumvents current code injection defenses such as W^X [25].

In the context of JIT compilers, JIT spraying [17] and similar attacks rely on the constants present in the source code emitted into the native code by the JIT compiler directly. The current state-of-the-art involves encrypting and decrypting these constants [14]. Though this avoids JIT spraying, however, adversaries can still make use of code-reuse techniques. An attacker can easily make use of the ubiquitous JIT compilers to generate new binary code having the necessary gadgets to make a malicious attack successful. The predictability of the native code produced by the JIT compiler, which fundamentally uses executable data, becomes a liability for developers, and an advantage for an adversary, that has been demonstrated in several attacks on Java run-time environments [13,23]. These kinds of attacks can lead to overwriting of the stack without actually going beyond the bounds of the data structures involved, and rely on the leakage of address space information as mentioned earlier. To thwart this leakage, we can leverage software diversity.

Software diversification involves the production of functionally identical, but syntactically different binaries of the same software. A simple, but effective form is code layout randomization. This raises the bar for ROP and return-into-libc attacks. Similarly, software diversity can prevent collusion attacks that identify new functionality or security fixes in updates [22]. Code layout randomization introduces uncertainty in the binary and hence, it drives up the complexity of attacks on diversified binaries.

Diversification can be introduced at various levels of granularity: instructions, basic blocks, loops, functions, programs, and finally systems. Techniques include NOP insertion, instruction randomization, stack layout randomization, function permutation, block reordering and splitting, heap randomization and so on [12].

A simple NOP insertion is every effective in the face of a ROP attacks, since its insertion shifts the address space information, and being completely randomized, it makes it difficult for attackers to chain the gadgets and deploy identical exploits.

The code layout is randomized by the compiler [2,3,4], using virtual machines [5] or through static binary rewriting [6,7]. However, most such defense techniques rely on static compilers or statically generated binaries. Homescu et al. [8] introduced the first concept of JIT diversification to support diversification for dynamically generated code. Our work is built upon their concept and extends it further to support adaptive live dynamic diversification. Whereas Homescu et al. randomize code only once, thus potentially still allowing attacks based on first leaking addresses and then exploiting those addresses, we close this hole by randomizing the code frequently during the program's execution.

3. DESIGN

The framework of the adaptive JIT diversification consists of two components: First, it builds on profile information generation and hot/cold method classification. Secondly, it inserts NOPs adaptively. From a security point of view, it suffices to inserts NOPs randomly. However, inserting them completely randomly, i.e., as frequently on hot code paths as on cold code paths, will introduce significant performance overhead. To avoid this, we want to concentrate more NOPs in cold code.

It is rather expensive, however, to collect fine-grained profile information to distinguish between hot and code cold blocks within methods already being JIT compiled based on coarse-grained, often sample-based information that the VM & JIT already collect by default. In fact, letting the JIT compiler insert execution counters to collect the profile information will typically be much more costly than introducing NOPs.

To prevent this, a key insight is that methods for which sufficient profile information is already available at some point in time, no longer need to be profiled. In other words, when methods have been recognized as hot enough, they no longer need to be profiled. From then on, when such a method is recompiled at random intervals to insert NOPs at new randomized places, the JIT compiler will insert no more execution counters.

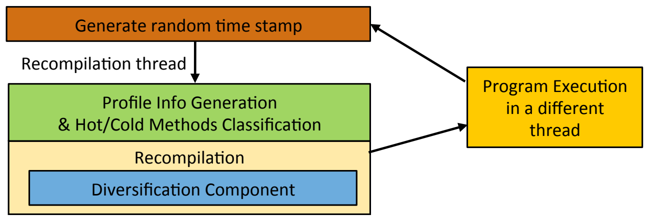
Generate random time stamp
Recompilation thread
Profile Info Generation & Hot/Cold Methods Classification
Recompilation
Diversification Component
Program Execution in a different thread

Fig. 1: Design Framework of Adaptive JIT Diversification

Fig. 1 shows a schematic diagram of the design. With the program executing, profile information is generated at random timestamps and hot and cold methods are classified. The recompilation component has a diversification component in it that performs adaptive NOP insertion. Note that the recompilation step does not stop the world. The recompilation occurs in a separate recompilation thread while the program is still executing. This further helps in reducing overheads, and makes the design useful for server-side scenarios, where programs are running indefinitely and one doesn't want to deploy stop-the-world techniques to apply diversification frequently. The recompilation process is cyclic and happens at time random timestamps.

3.1 Classification of Hot and Cold Methods

Basic blocks execution counting takes 5 instructions on x86: 2 for spilling and restoring a register, 2 for loading and storing a block's counter value, and 1 for incrementing it. The 4 memory operations cause a lot of overhead. Therefore, it makes sense to remove the counters as soon as possible. But of course that itself requires the generation of profile information.

Once we are able to "decide" that some method is hot, we have ample information, and hence we can at least remove the counters for that method's basic blocks. But how do we decide if a method is hot or cold? To do this, we defined a threshold of execution counts. If the execution count overshoots this threshold value, the method is classified as hot, else it is considered cold.

Also, readers must note that the order of recompilation of methods is important. The hot methods are recompiled first, followed by the cold methods, while the program is executing. This order of recompilation will lead to better adaptive diversification, as well as enhanced security and performance. The latter is explained in Fig. 2. Suppose method C is hotter than B, which in turn is hotter than A. So method C will be classified as hot first. Suppose that has happened some time before t_i, while the other methods are not (yet) classified as hot. In the compilation order of case 1, the execution counters in C will be omitted starting at time t_i. In the order of case 2, they will only be omitted at time t_j. By recompiling the hottest methods first, we hence loose less time.

Though sorting itself can involve significant overheads, experiments showed that we achieved a slight improvement overall because the sorting helped in getting the classification done quickly.

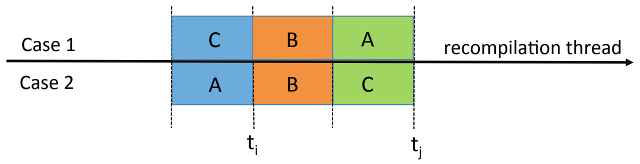

Fig. 2: Order of the methods' compilation

3.2 Adaptive NOP Insertion

We use a similar technique of adaptive NOP insertion as Homescu et al. [4]. A NOP normally takes 1 CPU cycle. Hence, if NOP insertion occurs in a basic block that executes a high number of times, it leads to high performance overhead. To deal with this, the idea is to insert less NOPs in hot basic blocks and relatively more NOPs in cold blocks. As compared to a single probability of NOP insertion, we replace this with a range of probabilities. The hottest block eventually gets the lowest probability of NOP insertion, and the coldest block gets the highest probability of NOP insertion. The intermediate blocks get their NOP insertion probability based on the following logarithmic function:

$$p_{NOP}(x) = p_{\max} - (p_{\max} - p_{\min}) \frac{\log(1+x)}{\log(1+x_{\max})} \quad (1)$$

where x is the execution count of the current basic block, x_{max} is the maximum execution count in the program and [p_{min}, p_{max}] is the probability range. For more details, we refer to [4].
Our randomized NOP insertion algorithm then is the following**:**

```
numNOPs = |NOP Table|
for i ∈ InstList do
  gen_rand = random(0.0,1.0)
  if gen_rand < pNOP(count(i)) then
    whichNOP = random (0, numNOPs)
    insert(i, NOP Table [whichNOP])
```

The x86 architecture provides 1 to 9 byte-sized NOP codes (Table 1), we also use 6 more variants of NOP (Table 2). These 6 variants preserve the processor state, while minimizing the probability of creating new gadgets, as the second byte decodes to an unusable operand or opcode. For instance, IN (read from an input port) requires privileged mode, and hence renders unprivileged code to return faults.

Table 1: 1-9 byte sized NOP instructions

Instruction	Encoding
NOP	90
66 NOP	66 90
NOP DWORD ptr [EAX]	0f 1f 00
NOP DWORD ptr [EAX + 00H]	0f 1f 40 00
NOP DWORD ptr [EAX + EAX*1 + 00H]	0f 1f 44 00 00
66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0f 1f 44 00 00
NOP DWORD ptr [EAX + 00000000H]	0f 1f 80 00 00 00 00
NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0f 1f 84 00 00 00 00 00
66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0f 1f 84 00 00 00 00 00

Table 2: Extra NOP variants

Instruction	Encoding	Second Byte Decoding
MOV ESP, ESP	89 E4	IN
MOV EBP, EBP	89 ED	IN
LEA ESI, [ESI]	8D 36	SS:
LEA EDI, [EDI]	8D 3F	ASS
XCHG ESP, ESP	87 E4	IN
XCHG EBP, EBP	87 ED	IN

3.3 Adaptive JIT Diversification

The following recompilation algorithm runs in a separate recompilation thread:

```
while True do
  sleep for random(0.0,1.0)*sleepTime
  sort all methods based on the execution counts
  for every method in all methods do
    if method.count > threshold the
      method.insertBBExecutionCount=False
    compile method in recompilation thread
```

This algorithm performs the following steps:

1. Sleep for random amount of time to randomize diversification timestamps.
2. Sort the unclassified methods in order of execution counts.
3. Compile them in the sorted order, inserting NOPs while doing so using the profile information as discussed above.
4. If execution counts exceeds threshold value, classify the method as hot, and remove its execution counters.

4. EVALUATION

We implemented our concept the Jikes Research Virtual Machine [24]. JikesRVM contains a baseline compiler, which is a fast compiler to convert bytecode into native code. We implemented our NOP insertion algorithm in this phase. We intend to adapt our solution to the optimizing compiler in our future work.

The diversifying JIT compiler was tested using the DaCapo 2009 Benchmarks [9] on a system containing an Intel Core i7-3610QM CPU @ 2.30GHz with HyperThreading enabled and 6GB RAM running Fedora 17 with Linux Kernel 3.6.11-5.fc17.x86_64. Table 3 presents the parameters and compiler options we added to the JIT compiler used in the experiment.

Table 3: JIT Compiler options added to JikesRVM

pNOP	Probability of NOP insertion in NOP randomization
pMax	Maximum probability of adaptive NOP insertion
pMin	Minimum probability of adaptive NOP insertion
profile_edgecounters	Enable or Disable Basic Block Execution Counters
sleepTime	Maximum interval time limit between two recompilations
Threshold	Threshold value of Hot/Cold classification
startTime	Time before starting recompilation for the first time
enableRecompile	Enable/Disable Recompilation

We evaluated three version of JIT diversification:

- V1: Random NOP insertion in one-time JIT diversification
- V2: Random NOP insertion in dynamic JIT diversification
- V3: Adaptive NOP insertion in dynamic JIT diversification

It is important to note that in this paper, we only present a performance evaluation of the proposed approach. The security evaluation for no-op insertion has been studied extensively in literature [2, 4, 8].

We varied the NOP insertion probability [p_{min}, p_{max}] between 0-30%, 10-50% and 25-50% for dynamic JIT diversification for versions 2 and 3, while for version 1, we fixed p_{NOP} at 30% and 50%. We varied the random sleep time and threshold values as follows:

sleep time = {500, 1000, 1500, 2000}
threshold = {100, 500, 1000, 1500, 2000}

To find the best configuration for our method for benchmark performance evaluation, we averaged out the performance overheads for each configuration. To do that, we first fixed p_{min} and p_{max} as 25% and 50% respectively, and varied the sleep time and threshold parameters, keeping one fixed and varying the other. For a sleep time value of 1000 and thresholds varying over the

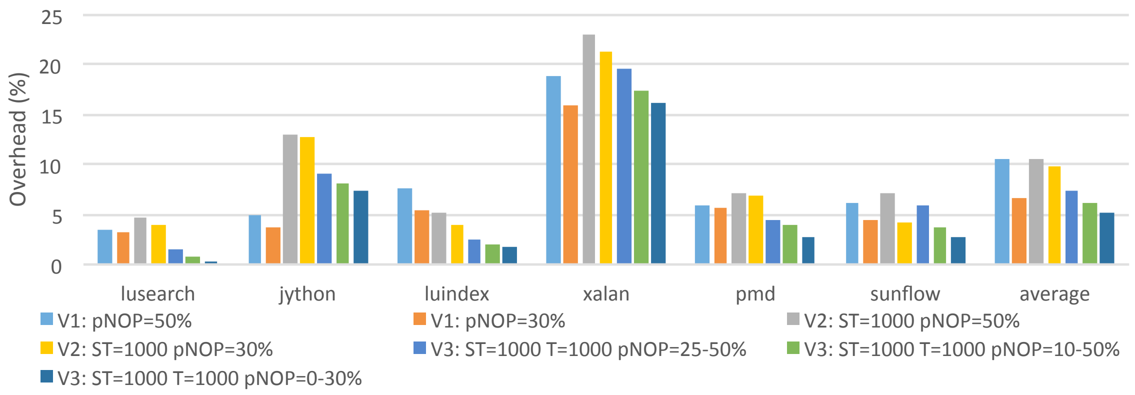

Fig. 3: Performance overheads comparison of best results obtained from all three versions evaluated

interval [100, 2000], we observed that all benchmarks showed an inverted bell shaped performance overhead curve, i.e., the performance overhead first decreases and then increases. We observed similar performance overhead patterns for sleep time = 1500 except for lusearch which showed strictly decreasing curve. There is a considerable change in pattern from sleep time = 500 to sleep time = 2000, where the bell shape pattern gets reversed for xalan, pmd and sunflow, while jython and luindex exhibited similar patterns of performance overheads. From these observations, we computed the sleep time and threshold values where minimal overhead was incurred. The best values achieved were sleep time = 1000 and threshold = 1000 for a minimal average overhead of 7.44%.

We used Eq. (1) to insert the p_{NOP} in the adaptive configuration of version 3. For version 2, we take the value of sleep time to be 1000 because this the value of sleep time for the best configuration in the adaptive system of version 3. The initial p_{NOP} for the adaptive version 3, i.e., the p_{NOP} value used for the first compilation when code is not yet known to be hot or cold, was set to 0.1.

Overall results are shown in Fig 3, in which ST stands for sleep time, and T for threshold. The performance overheads for version 1 are 6.7% and 5.3% for p_{NOP} = 50% and p_{NOP} = 30% respectively. Although version 2 increases this overhead to 10% and 9%, version 3 decreases it to 6% and 5% for p_{NOP} = 10-50% and p_{NOP} = 0-30% respectively, a reduction of 2x times as compared to version 2. It should be noted that the runtime overhead may look high for certain benchmarks like xalan. However that is the upper limit as observed in extreme cases. Also, the overhead in xalan in case of version 3 is comparable to version 2 (~16%). Except for jython, observe that adaptive no-op insertion in dynamic JIT diversification for p_{NOP} = 0-30% and sleep time and threshold values of 1000, provides the best performance results among all versions. In jython, version 1 incurs the least overhead. In all other cases, we achieved significant performance improvement with version 3, our approach, as compared to version 1 and 2.

5. RELATED WORK

Various methods have been proposed to harden JIT compilers against code reuse attacks. In this section, we briefly discuss these methods and approaches proposed.

JITSafe [13] applies immediate value elimination and obfuscation to protect against JIT Spray attacks. It reduces the time window of the JIT compiled code in the executable pages, applies immediate value elimination, followed by obfuscation of JIT compiled code. The evaluation by the authors show no false positives, while incurring upon low performance overhead.

In JITDefender [14], the code pages are marked non-executable at the code compilation point. Shortly before the code execution point, the pages are marked executable, and shortly after again, they are marked back as non-executable. Now, if the attacker hijacks the control flow and tries to perform JIT spraying attack, the access will be blocked since the VM keeps the code pages non-executable.

INSert, proposed by Wei et al [10] randomized register assignment, randomly transforms all immediate operands, parameters and local variables. It also randomly injects trapping snippets into the target code to alert the user of intrusion (JIT spraying) besides adding randomization as well.

Yee at al [11] introduced Native Client, a sandboxed execution environment, uses fault isolation and a secure runtime. NaCl manages the interfaces through which the system interaction and side effects are directed. NaCl originally did not allow dynamic code generation. However, this was later introduced in [16] by providing a more flexible form of software fault isolation.

Fine-grained address space layout randomization has been proposed to thwart runtime attacks. However, Snow et al [20] demonstrated just-in-time code reuse strategy that circumvents fine grained ASLR by frequently corrupting a memory disclosure to map an application's memory layout on-the-fly while also discovering gadgets at runtime. Thus a JIT based attack renders ASLR useless.

librando [8] is the first comprehensive work on hardening JIT compilers using concept of software diversity. It provides transparent code randomization in JIT compilers. It hooks itself to the the memory protection areas of the OS under consideration and randomizes newly generated code on the fly, while still preserving the calling stack's contents. Our work is very much based on the grounds of this work.

Niu and Tan proposed a control flow integrity (CFI) approach to harden JIT compilers, called RockJIT [19]. RockJIT is built upon modular control flow integrity. RockJIT build control flow graph from the JIT compiler's source and updates the control-flow policy of the JIT compiler dynamically when new code is generated on the fly.

Chobham [21] adopts RockJIT [19] to secure the browser's code and the JIT-compiled code since enforcement of control flow integrity (CFI) makes ROP gadget chaining difficult. Build on grounds of RockJIT, Chobam further constitutes three methods of further hardening of JIT compilers: first, it to improve the precision of the control flow graph generated by RockJIT, it deploys input triggered CGF generation; second, randomizing the order in which the callee-saved registers are restored; and third, it allocates a completely separate heap zone for all critical Javascript objects and adding checks to bound the access to these objects.

Isomeron [20] combines fine-grained code randomization with execution path randomization to mitigate typical ROP and JIT-ROP attacks. The authors further showed that Isomeron exponentially reduces the probability of the attacker to predict the correct runtime address of a target ROP gadget. Even usage of a randomization offset by a single byte reduces the attack success rate to 50%, even though it provides low entropy.

6. CONCLUSIONS AND FUTURE WORK

We presented a technique for adaptive just-in-time code diversification while re-randomizing binaries at random time stamps. This makes it much harder for adversaries to gain useful layout knowledge from leaked information and to exploit that knowledge.

With our experiments, we were able to diversify and re-diversify programs dynamically and at random timestamps, making use of collected profile information as we learn. Still our approach incurs small overheads, as it performs just as well as one-time diversification.

We currently used the baseline compiler in JikesRVM to implement our techniques. As a part of our future work, we intend to port the implementation to its advanced optimization system and compiler as well. This will allow us to study the overhead of the presented technique on fully optimized, hot code. At the same time, we will be able to leverage the profile information implicitly available when hot code is recompiled by the advanced optimization system.

In addition, we intend to research additional forms of diversification, such as basic block reordering and layout, and memory layout randomization.

Acknowledgment

Part of this research was funded by the Fund for Scientific Research - Flanders (FWO) under project grant 3G013013.

7. REFERENCES

[1] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Proc. ACM CCS, p. 552–561, 2007.

[2] T. Jackson, et al. Compiler-generated software diversity. Moving Target Defense, volume 54 of Advances in Information Security, pages 77–98. Springer New York, 2011.

[3] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In Proc. USENIX Security Symp, p. 475–490, 2012

[4] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler and M. Franz. Profile-guided automated software diversity. In Proc. CGO, p. 1–11, 2013.

[5] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In Proc. IEEE S&P, p.571–585, 2012.

[6] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In Proc. IEEE S&P, p. 601–615, 2012.

[7] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In Proc. ACM CCS, p. 157–168, 2012.

[8] A. Homescu, S. Brunthaler, P. Larsen and M. Franz. librando: Transparent Code Randomization for Just-in-Time Compilers. In Proc. ACM CCS, p. 993–1004, 2013

[9] Blackburn, S. M, et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis, Proc. ACM OOPSLA, p. 169–190, 2006.

[10] T. Wei, T. Wang, L. Duan, and J. Luo. INSeRT: Protect dynamic code generation against spraying. In Proc. ICIST, p. 323–328, 2011.

[11] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In Proc. IEEE S&P, p. 79–93, 2009.

[12] P. Larsen, S. Brunthaler, M. Franz. SoK: Automatic Software Diversity. In Proc. IEEE S&P, p. 276–291, 2014.

[13] Chen, P., Wu, R., Mao, B. JITSafe: a framework against Just-in-time spraying attacks. IET Information Security, 7(4):283-292, 2013.

[14] P. Chen, Y. Fang, B. Mao, and L. Xie. JITDefender: A defense against JIT spraying attacks. In Proc. IFIP TC SEC, p. 142–153, 2011.

[15] F. Cohen. Operating system protection through program evolution. Computers and Security, 12(6):565–584, 1993.

[16] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In Proc. ACM CCS, p. 298–307, 2004.

[17] D. Blazakis. Interpreter exploitation. In Proc. USENIX Workshop on Offensive technologies (WOOT), 2010.

[18] T. Bletsch, X. Jiang, V. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In Proc. ACM ASIACCS, p. 30–40, 2011

[19] B. Niu, G. Tan. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In Proc. ACM CCS, p. 1317–1328, 2014.

[20] L. Davi, C. Liebchen, A. Sadeghi, K.Z. Snow, F. Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return Oriented-Programming. In Proc. NDSS, p. 1–15, 2015

[21] B. Niu, G. Tan. Chobham: Taming JIT-ROP Attacks (Poster). In Proc. NDSS, 2015.

[22] B. Coppens, B. De Sutter, K. De Bosschere. Protecting Your Software Updates. IEEE Security and Privacy, 11(2):47–54, 2013.

[23] Java 7 SE Memory Corruption. Pwn2Own 2013. Accuvant Labs. Online whitepaper.

[24] B. Alpern et al. The JikesRVM Project: Building an open source research community. IBM System Journal, 44(2):399–418, 2005.

[25] PaX non-executable pages design & implementation. http://pax.grsecurity.net/docs/noexec.txt. (2004)