RandHeap: Heap Randomization for mitigating Heap Spray Attacks in Virtual Machines

Abhinav Jangda
University of Massachusetts Amherst, United States
aabhinav@cs.umass.edu

Mohit Mishra
ZotOut Inc.
mohit@zotout.com

***Abstract*—Virtual machines are an integral component of our present software systems infrastructure, including the web, and are here to stay. Web browsers like Google Chrome and Mozilla Firefox uses virtual machines to execute JavaScript code. Java Virtual Machines (JVMs) use just-in-time compilers to compile Java byte code to machine code. However, with the increasing use of virtual machines, they are also susceptible to security attacks. One such class of attack is the heap spray attack, wherein the attacker populates the heap with malicious code and exploits a vulnerability to jump to the populated malicious code in the heap, thereby enabling arbitrary code execution. In this paper, we present `RandHeap`, a technique to randomize the heap layout to detect and prevent heap spray attacks. `RandHeap` randomizes the heap in three different ways: (i) by randomizing object layout, (ii) by randomizing array layout, and (iii) by encrypting data stored on the heap. Using `RandHeap`, we were able to detect and prevent several heap spray attacks.**

For the evaluation of `RandHeap`, we implemented the concept of `RandHeap` in Google V8 and JikesRVM. We executed Octane 2.0 Benchmarks on Google V8 and Dacapo 9.12 Benchmarks on JikesRVM. Observations show that heap randomization using `RandHeap` is accompanied with low overhead and modest memory requirement. We implemented heap spraying attacks in Google V8 and JikesRVM and found that `RandHeap` was able to detect and prevent the attacks successfully.

I. Introduction

Today's infrastructure depends highly on virtual machines. Virtual machines are not only used by servers, but also by web browsers to execute JavaScript code. There has been the development of a very rich environment within virtual machines and web browsers. Unfortunately, this rich environment has also lead to numerous security issues. Virtual machines are often written in C/C++ exposing it to various vulnerabilities that can occur in programs written in these languages, such as buffer overflow, dangling pointer dereference and double free. These vulnerabilities are then exploited by attackers to execute malicious code on a virtual machine.

Heap spray attack [2] is one such attack to compromise web browsers by facilitating arbitrary code execution. It involves allocating a number of objects containing the exploit code in the heap, followed by triggering an existing vulnerability to execute the malicious code in the heap. Note that a heap spray attack does not trigger an attack by itself, but requires an existing memory vulnerability, like dangling pointer, buffer overflow, and double frees, to trigger an attack.

Current techniques [21][12][19][11] of preventing and detecting heap spray attacks suffer from one or more issues. Nozzle [21] examines heap at specific time intervals only. Hence, Nozzle can suffer from TOCTOU-vulnerability (Time-Of-Check-Time-Of-Use [6]), which means that an attacker can allocate a benign object, wait for Nozzle to examine it, then change it to contain malicious content, and trigger the attack. Also, Nozzle only examines a part of the heap at a time, and not the complete heap at once, to save on performance, leading to relatively less security impact. BuBBle [12] adds random padding between random positions in a string only because it makes an assumption that the malicious code will likely be stored as a part of a string only. BuBBle cannot not work if the attacker uses integer or float arrays, or objects to store the malicious code. DieHarder [19] is unable to prevent and detect heap spray attacks in virtual machines. Heap Taichi [11] describes four types of attacks and a technique to circumvent those attacks only. Attackers can extend these attacks to create more sophisticated attacks. We discuss the limitations of these works in detail in Section III.

In this paper, we first present the limitations of current techniques in protecting virtual machines from heap spray attacks. We then present `RandHeap`, a technique to prevent and detect heap spray attacks. Design of `RandHeap` is based on three principles: (i) encrypting heap data, (ii) randomizing heap layout, and (iii) low performance overhead with low memory requirements. `RandHeap` randomizes the heap layout using three techniques:

1) Whenever a new class is loaded, a random layout is decided for all the objects instantiated from the loaded class. A random layout is decided by randomizing the field locations and adding random padding between the fields.
2) An array layout is created during the virtual machine's boot phase. The array layout is created by randomizing the element locations and adding random padding between different elements. All the elements in any array will be accessed based on this array layout.
3) Encrypting heap data to prevent malicious code stored on heap to create a valid instruction sequence.

The techniques mentioned above prevent heap spray attacks by: (i) making it harder for an attacker to guess the location of data stored in the heap, thereby making it difficult to store malicious code on heap in a well-defined or an ordered manner, and (ii) preventing the correct execution of malicious code by encrypting the data stored on heap. By setting the

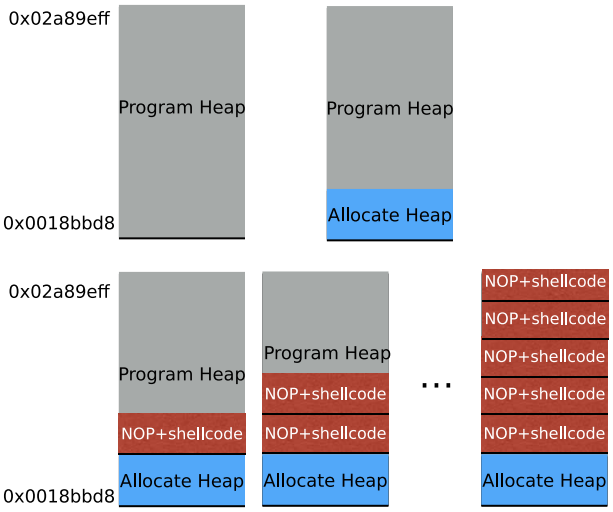

Fig. 1: Program Heap and flow of heap spraying

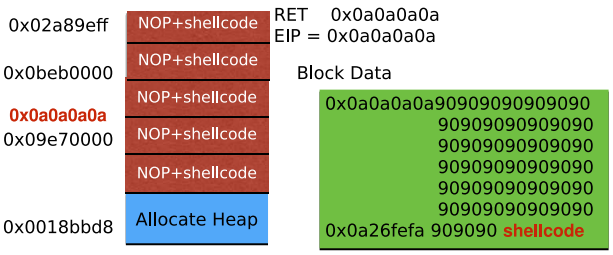

Fig. 2: Arbitrary code execution on the heap via successful return to `0x0a0a0a0a`

value of random padding to the `call` instruction to call a function, we ensure that during a heap spray attack, this function is called and `RandHeap` is able to detect the attack.

a) Contributions: This paper makes the following contributions:

- We present limitations of current techniques used for protecting heap spray attacks (Section III).
- We introduce techniques to perform the following:
  - randomize object layout (Section IV-A1).
  - randomize array layout (Section IV-A2).
  - encryption of data stored on the heap (Section IV-A3).
- We integrated `RandHeap` in Google V8 [1] and JikesRVM [8]. We then report the results of performance evaluation and memory requirement (Section VI). We show that we successfully protect:
  - JikesRVM with average performance overhead of 6% with on an average 5-10% extra memory requirement.
  - Google V8 with average performance overhead of 6% with on an average 6-9% extra memory requirement.

II. Threat Model

In this section we present our assumptions about the defenses the system employs, capability of the attacker, and presence of memory related bugs in the system.

We make the following assumptions for the threat model used in the research:

1) the attacker has sufficient capability to launch repeated attacks and allocate objects on the heap repeatedly.
2) the attacker has the limitless ability to allocate and free objects on the heap at will.
3) the victim's system contains at least one exploitable security vulnerability to point the instruction register (e.g. `EIP` in x86) to an address on the heap.

Provided below is a simple explanation of how a typical heap spray attack ideally works. Figures 1 and 2 depict the flow of heap spray attack. We assume a buffer overflow vulnerability exists in the program.

Consider the following code snippet:

```
HeapBlock = new Array();
for (int i = 0; i < heapBlocks; i++) {
    HeapBlock += NOPSled + Shellcode
}
```

The code snippet sprays the heap and allocates HeapBlock one on top of the other `heapBlocks` number of times. Each HeapBlock is a `NOP` sled ending in a shellcode.

After heap spray, the buffer is filled with a return address, say `0x0a0a0a0a`. After returning, the EIP is set to `0x0a0a0a0a`. The code execution at this point has a heap block which eventually leads to the shellcode execution.

III. Limitations of the State-of-the-art

In this section, we describe the overview and limitations of current techniques.

A. Nozzle and Heap Taichi

Nozzle [21] works by creating a control flow graph of the heap at specified time intervals and then analyzing this control flow graph to detect `NOP` sled and shellcode. Since, Nozzle analyze heap only at specified time intervals, it suffers from TOCTOU-vulnerability (Time-Of-Check-Time-Of-Use [6]). To exploit this vulnerability, an attacker can allocate a benign object, wait for Nozzle to examine it, then change it to contain malicious content, and trigger the attack. Also, Nozzle only examines a part of the heap at a time to save on performance, leading to less security.

Heap Taichi [11] describes new attacks, which do not require `NOP` sled and hence, decreases the surface area of the attack. The defenses against these attacks, described in [11], are implemented on the top of Nozzle. Hence, these defenses also suffer from the same issues as Nozzle except the low surface area of attack.

B. BuBBle

BuBBle [12] provides security against heap spray attacks by adding random padding before storing strings on the heap. Before reading a string, BuBBle returns the original string by removing the random padding. Hence, BuBBle only provides protection from heap spray attacks if the `NOP` sled

and shellcode are stored in the form of a string. On the other hand, if the NOP sled and shellcode are stored in the form of integer arrays or objects, BuBBle will be unable to provide protection against heap spray attacks. Since, BuBBle traverses the whole string to remove or add random padding before accessing or storing a string, BuBBle would perform inefficiently while accessing a large string.

C. DieHarder

DieHarder [19] is based on DieHard, which is a high performance memory allocator and allows a program with memory errors to execute correctly with a high probability. DieHard uses a bitmap-based fully-randomized memory allocator and allocates object at random places on the heap. On top of DieHard, DieHarder [19] provides security against heap-based attacks by using Sparse Page Layout, where small objects are allocated within pages spread sparsely across the address space and using Address Space Sizing, which randomizes the addresses of small object pages.

Although DieHarder provides security against heap spray attacks for native executables, we found that this is not true for managed languages. We wrote a heap spray attack in JavaScript, which sprays the heap with NOP sled and shellcode. The JavaScript file calls a C++ function which contains a buffer overflow vulnerability. We used this vulnerability to set the return pointer to an arbitrary location on the heap. This JavaScript file was executed on Google V8. Interestingly, even with DieHarder enabled, the shellcode got successfully executed.

We believe the reason for this limitation of DieHarder is, unlike C/C++ programs, which allocates memory in small fragments, virtual machines usually allocate large chunks of the heap. Objects are allocated from this large chunk, leading to a decreased amount of randomization and successful prediction of the location of shellcode.

IV. RandHeap: Design and Analysis

RandHeap provides protection against heap spray attacks by (i) randomizing the object and array layouts, and (ii) encrypting data on the heap. RandHeap consists of two essential components: (i) Attack Prevention, and (ii) Attack Detection. Attack prevention is achieved by randomizing the objects' and the arrays' layout and data on the heap. Attack detection is accomplished by injecting a binary code for call instruction, thereby, calling the specified function in case of a heap spray attack.

A. Attack Prevention

The Attack Prevention component deals with preventing the attacker from deploying a successful heap spray attack. This component randomizes the location of fields of all objects, elements in an array, and encrypting the data stored on the heap. By randomizing the heap and encrypting the heap data, we make sure that (i) the malicious code is not stored in a contiguous manner and/or at attacker defined locations, and (ii) the malicious code stored on the heap in the encrypted form would not form a valid instruction sequence. This helps in preventing the machine from executing a malicious code, as demonstrated in Section V.

1) Randomizing Object Layout: When the virtual machine is resolving a class, its methods and fields, the attack prevention component will randomize the layout for every class. This randomized layout is used by every instantiated object of this class. The layout is randomized by randomizing the locations of fields of the class and adding random padding of paddingSize bytes between each field. As a result, the fields are now updated to new memory locations. The JIT Compiler compiles the code to access the fields using the new layout.

Algorithm 1 Object Layout Randomization Algorithm

Require: The given object layout for a given class in $objectLayout$ variable
Ensure: Randomized new object layout represented as $newLayout$

```
1:  function ObjectLayoutRandomization(objectLayout, paddingSize)
2:      newObjectLayout ← φ
3:      boolArray ← new boolean [objectLayout.length()]
4:      for i = 0 → boolArray.length() do
5:          boolArray[i] ← false
6:      for i = 0 → objectLayout.length() do
7:          j ← rand() % objectLayout.length()
8:          while boolArray[j] = true do
9:              j ← rand() % objectLayout.length()
10:         newObjectLayout.add (objectLayout[j])
11:         boolArray[j] ← true
12:     for i = 0 → objectLayout.length() do
13:         j ← paddingSize×(rand() % 2)
14:         if j ≠ 0 then
15:             for k = 0 → j do
16:                 newObjectLayout.insert (i, new ByteField())
        return newLayout
```

Algorithm 1 presents the Object Layout Randomization Algorithm. The procedure $ObjectLayoutRandomization$ takes $objectLayout$ and paddingSize as arguments. Argument $objectLayout$ represents the object layout, i.e., the placement of fields in an object of a class and paddingSize is the size (in bytes) of each padding. $ObjectLayoutRandomization$ returns the randomized object layout. Line 2 allocates $newObjectLayout$ and initializes it to ϕ. Lines 4-5 allocates a boolean array and initializes all its elements to false. Lines 6-11 takes a random field in each iteration from $objectLayout$ and add it to $newObjectLayout$. Lines 12-16 creates a series of $ByteField$ based on a random number and add them to $newObjectLayout$.

Figure 3 shows a class Example with three fields and paddingSize = 4. We assume that the size of the address of an object is 4 bytes. With the original object layout, the fields are allocated one by one in a predefined manner. However, the fields have been allocated randomly in both randomized object

Algorithm 2 Array Element Access Algorithm

```
1: function ARRAYELEMENTACCESS(index, indexArray, indexArraySize, indexArrayLength)
2:     rem ← index % indexArrayLength
3:     quotient ← index ÷ indexArrayLength
4: return quotient × indexArraySize + indexArray[rem]
```

layouts. For example, in the first randomized object layout, the order of the fields is `f2`, `f3`, and `f1`. Further, random padding is inserted between fields.

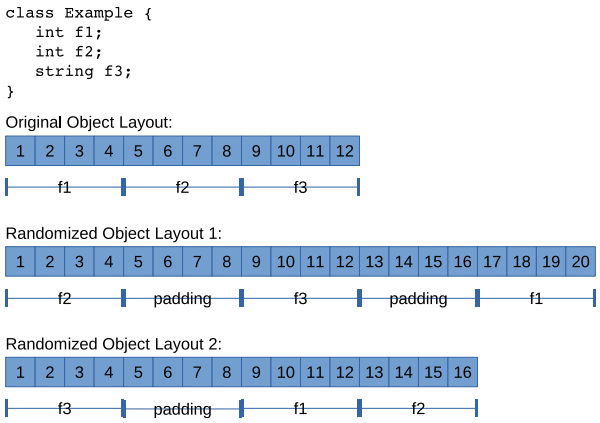

Fig. 3: Example class, original object layout and two randomized object layouts

2) Randomizing Array Layout: An array stores elements in contiguous memory locations. Hence, the complexity to index an array element is $O(1)$. The Attack Prevention component randomizes the array layout while maintaining this property of arrays. At the start of the virtual machine, the Attack Prevention component will allocate an array of integers of a length provided by the user. We denote this array as `indexArray` with the number of elements as `indexArrayLength`. `indexArray` defines a layout for all arrays. `indexArray` is a mapping between logical index to physical index. Logical index is the index of an element in the array, while physical index is the index where that element is stored in the memory. The number of elements of each allocated array is a multiple of `indexArraySize`. Algorithm 2 shows the algorithm to access an array element at an index. The *ArrayElementAccess* function takes four parameters: `index` of the element, `indexArray`, `indexArraySize`, and `indexArrayLength`. Line 2 and 3 finds the remainder and the quotient by dividing `index` with `indexArrayLength`. Line 4 returns the *physical index*.

Every time when the virtual machine starts, it will decide a random layout for arrays, i.e. construct `indexArray` based on the `indexArrayLength` given by the user. Deciding the random layout is based on two techniques: randomizing element locations and adding random padding. Consider an `indexArray` with elements: [4, 7, 5, 6, 8]. Since, the maximum physical index of an array is 8 and 8 is the position where the second last element ends (or the last element begins), the size of the array should be 8 + 1 = 9. Because padding is added in between two random indexes, the size of an array cannot always be a multiple of `indexArrayLength`. For this reason, every array is allocated as a multiple of `indexArraySize`, which is equal to the addition of maximum value in `indexArray` and 1. Algorithm 3 depicts array layout randomization algorithm. *ArrayLayoutRandomization* takes `paddingSize`, which is the size of each padding and `indexArrayLength` as parameters. This algorithm can be divided into the following phases:

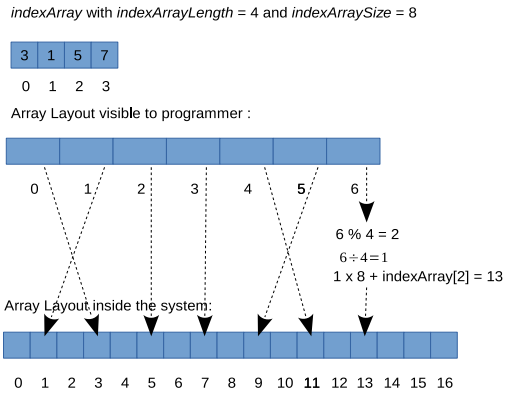

Fig. 4: Example array layout randomization

- The first phase allocates `indexArray` of length `indexArrayLength`. This process is performed at line 2.
- In the second phase, the values at each index in `indexArray` are set to random numbers less than `indexArrayLength`. Moreover, the use of `boolArray` ensures that the values at each index in `indexArray` are distinct. This process is responsible for randomizing element locations. This process is done at lines 3-11.
- In the third phase, for each index a random number is decided, which is then added to this index and all the indexes following it. This process is responsible for adding random padding of `paddingSize` bytes, and is performed at done at lines 12-16. At the end, `indexArraySize` is assigned the value equal to the max element in `indexArray` plus 1.

Figure 4 presents a randomized array layout for `paddingSize` = 1, `indexArray` with `indexArrayLength` = 4 and `indexArraySize` = 8. The layout defined by `indexArray` maps logical index 0 → physical index 3, logical index 1 → physical index 1, logical index 2 → physical index 5, and logical index 3 → physical index 7. Figure 4 also shows an example array of 6 character elements and mapping of logical indexes to physical indexes.

a) Optimization: Unfortunately, the above technique would yield high overhead because division, multiplication,

Algorithm 3 Array Layout Randomization Algorithm

```
1: procedure ARRAYLAYOUTRANDOMIZATION(paddingSize, indexArrayLength)
2:     indexArray ← new int[indexArrayLength]
3:     boolArray ← new boolean [indexArrayLength]
4:     for i = 0 → indexArrayLength do
5:         boolArray[i] ← false
6:     for i = 0 → indexArrayLength do
7:         j ← rand() % indexArrayLength
8:         while boolArray[j] = true do
9:             j ← rand() % indexArrayLength
10:        indexArray[i] ← j
11:        boolArray[j] ← true
12:    for i = 0 → indexArrayLength do
13:        k ← m×(rand() % 2)
14:        for j = i → indexArrayLength do
15:            indexArray[j] ← indexArray[j] + k
16:    indexArraySize ← max(indexArray) + 1
```

and addition are costly operations. To solve this problem we optimized the solution as follows:

- We ensure that `indexArrayLength` assigned by the user is a power of 2.
- We ensure that `indexArraySize` is also a random number which is a power 2 and is greater than `indexArrayLength`.
- For any two integers n and x, $x \ \% \ 2^n$ and $x \ \& \ (2^n - 1)$ yields the same result, i.e. the remainder of division when x is divided by 2^n. Hence, we decided to use bit-wise `and`(&) operator instead of binary `mod` (%) operator in Algorithm 2 to fetch the remainder.
- For any two integers n and x, $x \ \div \ 2^n$ and $x \ >> \ n$ yields the same result, i.e. the quotient of division when x is divided by 2^n. Hence, we decided to use shift left ($>>$) operator instead of binary `div` ($\div$) operator in Algorithm 2.
- For any three integers n, x, and y, $x \ \times \ 2^n + y$ and $(x \ << \ n) \mid y$ yields the same result. Hence, we decided to use the shift right ($<<$) and bit-wise `or` (—) operators instead of the combination of binary `mult` ($\times$) and binary `plus` ($+$) operators in Algorithm 2.

Algorithm 4 is the optimized array access algorithm. It takes 5 parameters: `index` of the element, `indexArray`, `indexArrayLength`, `logIndexArrayLength`, and `logIndexArraySize` as parameter, where `logIndexArraySize` is equal to $log_2(indexArraySize)$ and `logIndexArrayLength` is equal to $log_2(indexArrayLength)$

3) Encrypting Heap Data: In a heap spray attack, the attacker populates heap with the malicious code. This malicious code is then executed when the attacker directs the instruction pointer to a location in the heap. Instead of storing the data in its original form, `RandHeap` stores the data in an encrypted form and decrypts it before reading. When the virtual machine

Algorithm 4 Optimized Array Element Access Algorithm

```
1: function OPTIMIZEDARRAYELEMENTACCESS(index, indexArray, indexArrayLength, logIndexArrayLength, logIndexArraySize)
2:     rem ← index & (indexArrayLength-1)
3:     quotient ← index >> logIndexArrayLength
4: return (quotient << logIndexArraySize) — indexArray[rem]
```

is booted, it produces a random key, for example, by using the `rand` function. While storing data on the heap, i.e. setting a value to a field of an object or an element of an array, the data is first encrypted by `XOR`ing it with the key and then stored. When data from the heap is read, it is then decrypted by `XOR`ing it with the same key and then returned. The size of key is the maximum size of a data type supported by the language, which is 64 bits in Java and JavaScript.

Due to the encryption, the malicious code would not get executed as this encrypted form of code would not form the desired instruction sequence. Moreover, the randomized selection of key makes sure that the attacker can not bypass encryption by storing data, which when encrypted could produce malicious code.

B. Attack Detection

The Attack Detection component detects the heap spray attack, and contains the following two sub-components:

- Attack Detection Function: This function will be called as the attack commences. In our current implementation, this function calls the C Library's `abort` function.
- Assign value to random padding: The random padding inserted by the Attack Prevention component is of `paddingSize` bytes. This random padding will contain a `call` instruction and the function address for this call instruction is set to the Attack Detection Function. Hence, `paddingSize` must be at least equal to the number of bytes required to encode `call` instruction and the function address for this call instruction.

The working of the Attack Detection component is based on the fact that while the malicious code sprayed on the heap is executed, the instruction pointer will encounter the random padding, and therefore the `call` instruction will be executed.

V. EXAMPLE

In this section, we demonstrate the working of our technique. In this example, we assume that the heap is sprayed using a long string containing `NOP` sled and the shellcode. In this example, we assume an x86 32-bit CPU and fix `paddingSize` = 6. To make this example simple, suppose the random key is set to `0x90` and use a simple shellcode:

```
mov al, 1
xor ebx, ebx
int 0x80
```

The above shellcode calls `exit` system call with argument 1 in register `al`. When compiled, this assembly turns into the following machine code sequence:

```
b0 01 31 db cd 80
```

Each byte of the above machine code when XORed with `0x90` produce the following instruction sequence:

```
20 91 a1 4b 5d 10
```

In this example, the heap is first sprayed by `NOP` instructions followed by insertion of the shellcode. Figure 5 shows the heap layout after it is sprayed. Logically the array is visible to the programmer as a contiguous memory with the sprayed heap code. However, the actual layout is not contiguous; it is randomized and padded at random addresses with encrypted data. The padding contains the `call` instruction, which calls the attack detection function. In this example, the address of the attack detection function is `0x401214`. In the heap, the bytes of shellcode has been randomized and encrypted, hence, the shellcode is not in its correct form, which when executed would produce the required malicious attack.

When the attacker exploits a buffer overflow and transfers the position of the instruction pointer to a guessed location on the heap (as shown in Figure 5), the CPU starts executing the code. Eventually, it encounters the `call` instruction and transfers the control to the *attack detection function*.

Similar example can be given for how the object layout randomization helps in the prevention of attack. Consider the object `shellCode` formed by the following JavaScript code:

```
var shellCode = {f1: 0xb0019090,
                 f2: 0x31db9090,
                 f3: 0xcd809090}
```

The code above forms an object `shellCode` containing the bytes of shellcode as its fields. The first field encodes the bytes `b0`, `01`, and two NOP instructions with byte `90`. NOP instructions are added after one instruction to fill the padding between different elements. This object with `NOP`-sled can be sprayed on the heap. Following JavaScript code shows one of the object layouts and value of fields after object layout randomization and encryption of the heap data:

```
var shellCode = {f3: 0x5d100000,
                 f1: 0x20910000,
                 f4: 0xff151412400000,
                 f2: 0xa14b0000}
```

In the code above, the fields have been randomized and a random padding field `f4` with value equal to `call 0x401214` have been added. Since, the execution of `shellCode` will now start with field `f3` instead of `f1`, the shellcode will not produce the intended result. The control during the execution of the shellcode will eventually be transferred to the attack detection function due to the random padding field.

VI. Evaluation and Implementation

We implemented `RandHeap` in Google V8 5.5 [1] and JikesRVM 3.1.3 [8]. We performed three types of evaluations: (i) performance evaluation to measure execution time overhead of `RandHeap`, (ii) memory evaluation to determine extra memory required by `RandHeap`, and (iii) security evaluation to determine whether `RandHeap` can prevent heap spray attacks. For performance and memory evaluation of `RandHeap` implementation in Google V8, we executed Octane 2.0 Benchmarks [5] on two versions of Google V8 with Google V8's Generational Garbage Collector: (i) Google V8 without `RandHeap`, and (ii) Google V8 with `RandHeap`. Octane 2.0 Benchmark suite contains standard JavaScript benchmarks, which are designed to evaluate JavaScript engine's performance. For performance and memory evaluation of `RandHeap` implementation in JikesRVM, we executed Dacapo 9.12 Benchmark suite [9] on two versions of JikesRVM with JikesRVM's Mark and Sweep Garbage Collector: (i) JikesRVM without `RandHeap`, and (ii) JikesRVM with `RandHeap`. Dacapo 9.12 Benchmark suite is a standard benchmark tool to evaluate Java Virtual Machine's performance. We performed our evaluation on a platform containing Quad Core Intel i7-3610QM with HyperThreading disabled and each core running at 2.3 GHz with 6 GB RAM running 64-bit Fedora 21 with Linux Kernel 3.19.3. For both Google V8 and JikesRVM, heap is set to 1 GB. We took 4 values of `indexArrayLength`: 4, 8, 16, and 32. For each of these values, each benchmark was executed 5 times. We report average performance overhead and memory requirement of `RandHeap`. We are only reporting a subset of the Dacapo 9.12 benchmark suite since several benchmarks did not work on a vanilla JikesRVM running on our evaluation system. This is a known problem and unrelated to `RandHeap`.

A. Performance Evaluation

Figure 6 presents the performance overhead of `RandHeap` (in %) for Octane benchmarks executing on Google V8. For each value of the `indexArrayLength` average overhead is around 6% for Google V8. Although, few instructions are added for every element access in arrays (as given in Section IV-A2), caching of `indexArray` brings the overhead to an acceptable value. Since `indexArray` is small in size for each of the `indexArrayLength`, it can readily fit in L1 cache, thereby providing faster access from logical to physical indexes. Overhead above 5% for all values of `indexArrayLength` is observed in `zlib`, `Splay`, `DeltaBlue`, and `Navier Stokes`, because these benchmarks heavily uses array indexing operations and object creation and destructions. All other benchmarks with performance overhead less than or equal to 4% does not contains significant amount of property load/store, object, and array indexing operations, leading to low overhead.

Figure 7 presents the performance overhead of `RandHeap` (in %) for Dacapo benchmarks executing on JikesRVM. For every value of `indexArrayLength` the average overhead is around 6% for JikesRVM. We believe caching of `indexArray` is responsible for small overhead. Overhead of more than 6% is observed in `lusearch`, `luindex`, `xalan`,

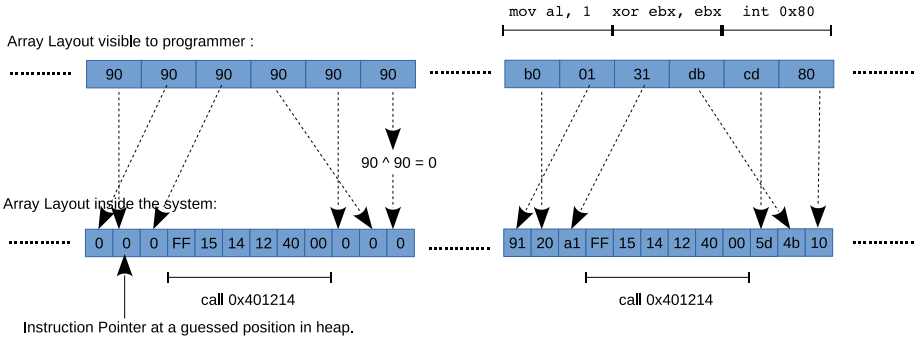

Fig. 5: Example attack: heap layout after heap spraying

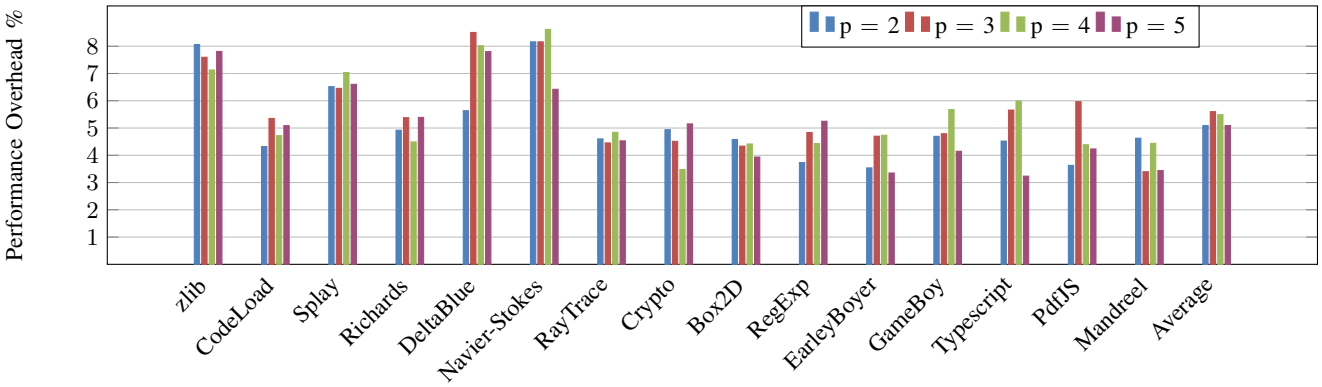

Fig. 6: Performance Overhead (in %) of Google V8 with `RandHeap` over Google V8 without `RandHeap`, where p = log_2(`indexArrayLength`)

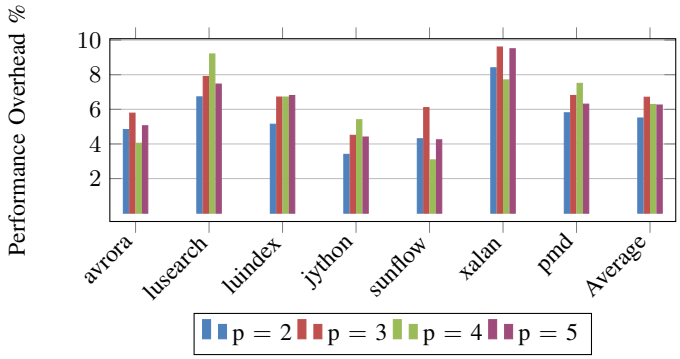

Fig. 7: Performance Overhead (in %) of JikesRVM with `RandHeap` over JikesRVM without `RandHeap`, where p = log_2(`indexArrayLength`)

and `pmd` because these benchmarks rely heavily on string indexing operations.

1) Comparison with related work: Nozzle [21] suffers from an average overhead of around 7%. Heap Taichi [11] suffers from an average overhead of 5%, when evaluated on Google V8 and Firefox. BuBBle [12] has an overhead of 2.6% on Google V8 benchmarks. DieHarder [19] suffers from an average overhead of 20% with the highest overhead of 117%. However, for Firefox, DieHarder is reported to have an no overhead. This is because a virtual machine do not allocate memory in small fragments like C/C++ programs instead, they allocate a large chunk of memory and objects are allocated memory from this large chunk leading to significantly less number of calls to DieHarder.

`RandHeap` suffers from an average overhead of 6%, which is less than all of the previous techniques except BuBBle. However, BuBBle only adds padding in strings not in objects or arrays, hence, provides protection if the shellcode is stored in the form of a string. Moreover, BuBBle traverses the whole string before writing to add padding and before reading to remove padding, which, if applied to all types of arrays could lead to a significant overhead. In other words, the complexity of array indexing is $O(n)$ in BuBBle. However, `RandHeap` provides array indexing operations with $O(1)$ complexity.

B. Memory Requirements

For both versions of Google V8 and JikesRVM, we calculated the total memory usage as the sum of the heap usage before every garbage collection cycle. Figure 9 shows the extra memory (in %) required by Google V8 with `RandHeap` over Google V8 without `RandHeap`. Figure 8 shows the extra memory (in %) required by JikesRVM with `RandHeap` over JikesRVM without `RandHeap`. We observe that the average memory requirement increases with increase in `indexArrayLength`. For Google V8 with

RandHeap, the average memory requirement is 6.74% for indexArrayLength = 8. For JikesRVM with RandHeap, the average memory requirement is 5.29%. Since, RandHeap increases the size of objects and arrays, benchmarks allocating significant number of arrays and objects are expected to have higher memory requirements. Memory requirement greater than 10% in Figure 9 is observed for zlib, CodeLoad, RegExp, GameBoy, PdfJS, and Mandreel because these benchmarks allocates significant number of strings and objects. Similarly, we observe higher memory requirement (greater than 10%) in Dacapo Benchmarks for avrora, sunflow, and pmd.

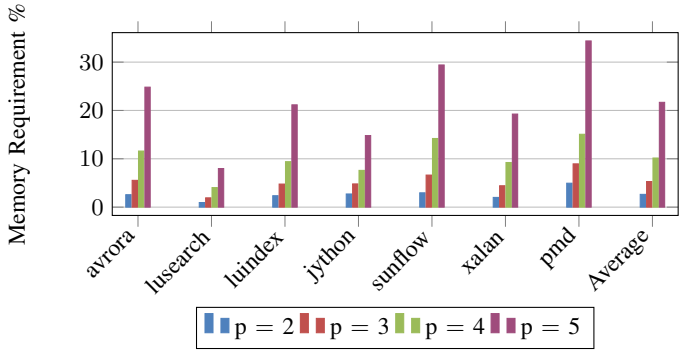

Fig. 8: Memory Requirement (in %) for JikesRVM, where p = log_2(indexArrayLength)

1) Comparison with related work: Nozzle, Heap Taichi, and DieHarder doesn't report memory requirements. BuBBle report memory requirement of 5.3%, which may appear to be less than RandHeap's memory requirement at the first sight but, please note that BuBBle adds random padding only in strings, while RandHeap adds random padding in all types of arrays and objects. Hence, we expect that if BuBBle is applied to all types of arrays, the memory requirement of BuBBle will be close to RandHeap's memory requirement.

C. Security Evaluation

Heap spray attack makes use of spraying objects in the heap in three ways: (i) using a string, (ii) using an array of integers or floats, and (iii) using an object with several byte fields containing shellcode. We created an attack by calling a C++ function in JavaScript in Google V8. This C++ function contains a buffer overflow, which can be exploited to set the return pointer to an arbitrary place in the heap. We created a similar attack in Java in JikesRVM, which calls a C++ function containing a buffer overflow.

In each of the above cases, RandHeap was able to detect the attack. The instruction pointer landed within the heap, and encountered random padding containing the call instruction. The CPU executed this call instruction to call attack detection function. At times when the instruction pointer directly landed into the sprayed shellcode, then due to the randomization of array elements the shellcode never got executed correctly. Moreover, random padding also prevented the attack by preventing to guess the padded bytes and setting a jump to shellcode.

Heap Taichi [11] described four types of objects using which another heap spray attack can be created. However, all these attacks also require that malicious code must be stored in a well defined way in the heap without random padding. Hence, RandHeap can also prevent from attacks. RandHeap was able to detect these heap spray attacks generated by spraying heap using each of the four types of objects.

1) Security Analysis: For the successful execution of any machine code, two conditions must be satisfied: (i) machine code instructions are executed in a correct manner, and (ii) the control flow must not be transferred somewhere else during the execution of the code. We demonstrate that for heap spray attacks, RandHeap prevents the successful execution of the shellcode by preventing the above conditions, thereby, preventing a malicious code to be executed via heap spray.

There are different ways to generate spray shellcode including the use of arrays, objects, and/or strings. The shellcode is preceded by NOP sled. During the heap spray attack, there are two situations depending on where the instruction pointer lands on the heap:

1) *Instruction Pointer lands on the NOP sled*: For the instruction pointer to reach the shellcode, it has to execute all NOP instructions. Since, NOP instruction does not modify the current state in a processor, randomizing the locations of NOP cannot prevent the attack. However, because of the random padding in between random locations in NOP sled, the call instruction will be executed, thereby calling attack detection function and detecting the attack.
2) *Instruction Pointer lands just before the shellcode*: For the shellcode to be executed successfully, conditions (i) and (ii) must be satisfied. However, on the heap, malicious code is stored in an encrypted form and the specified order defined by the attacker is not followed to store the shellcode on the heap but each instruction of the shellcode is stored at a random location. Moreover, padding in between random locations of the shellcode, contains a call instruction. Hence, the shellcode is not executed correctly due to randomized order of instructions, encryption of these instructions, and the execution of call instruction leads to the transfer of control out from the shellcode.

Hence, RandHeap can prevent and detect all kinds of heap spray attacks.

To guess the array layout correclty, an attacker has to precisely know the randomized location of the array elements and the location of the random padding. The possible number of element arrangements in indexArray are indexArrayLength!. The number of cases for padding is $2^{\texttt{indexArrayLength}}$. Hence, the number of possible array layouts is $\texttt{indexArrayLength}! \times 2^{\texttt{indexArrayLength}}$. Table I depicts the number of possible array layouts for different values of indexArrayLayout. For indexArrayLength = 16, there are 1.3×10^{18} possible

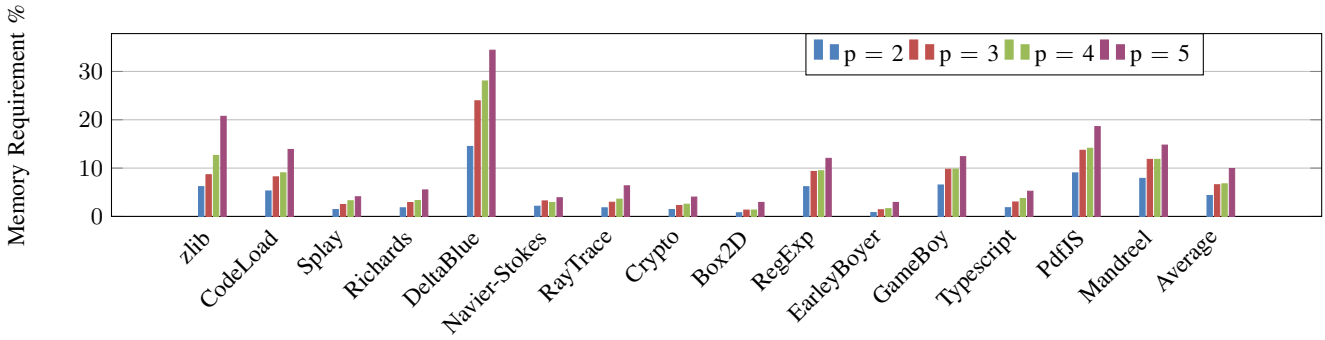

Fig. 9: Memory Requirement (in %) for Google V8, where p = log_2(`indexArrayLength`)

array layouts, hence, the probability is 7.7×10^{-19}. Since, the size of random key is 64 bits, there are 2^{64} number of possible keys. Therefore, the probability of guessing the correct array layout and random key is far too low to consider for any feasibility. Similarly, the number of object layouts for an object with `n` number of fields is $n! \times 2^n$.

2) Comparison with related work: Nozzle provides security by analyzing the heap at specified heap intervals and detecting `NOP` sled. Nozzle suffers from three major issues: (i) TOCTOU vulnerability [6], i.e., the attacker can allocate a benign object, wait for Nozzle to examine it, then change it to contain malicious content and trigger the attack, (ii) to decrease the performance overhead, Nozzle examines only a part of the heap at one time, (iii) Heap Taichi has shown that even without `NOP` sled, successful heap spray attacks can be created. Techniques [11] to prevent Heap Taichi based attacks are based on Nozzle and hence, these techniques also suffer from (i) and (ii) issues described above. Moreover, Ding et al. [11] claims that new attacks based on Heap Taichi can be developed, which are difficult for current techniques to detect. Since, `RandHeap` randomizes the heap layout and encrypts the heap data, it is able to prevent Heap Taichi based attacks and does not suffers from the above issues.

By adding random padding in between the strings, BuBBle [12] can prevent heap spray attacks, which stores shellcode in a string, but, BuBBle cannot prevent attacks which can store shellcode in integer or float arrays, and/or objects. However, `RandHeap` is able to detect and prevent these attacks because it randomizes objects and all types of arrays.

We have shown in Section III that DieHarder [19] is not able to protect managed languages from heap spray attacks. However, `RandHeap` does heap randomization at the virtual machine level and hence, is able to protect virtual machines.

VII. Related Work

Much of the related work has been described in Section III. In this section, we will describe other related works.

A. Memory Attack Prevention

A number of attacks makes misuse of C APIs, `strcpy` in particular. LibSafe [4] and HeapShield [3] libraries can prevent such attacks that exploit the C APIs. Buffer Overflow attacks mostly depend on the attacker injecting a malicious code in memory that is executable in nature. Data Execution Prevention (DEP) [23] is a Windows countermeasure to mitigate such memory corruption attacks by marking one or more pages in memory as non-executable. However, DEP alone is well known to be by-passable as presented in [23].

Some of the attacks work using the fact that certain processes are loaded at specific memory locations, i.e. the memory locations for certain programs are always static. Address Space Layout Randomization (ASLR) [18] randomizes memory locations used by system files and other programs. However, even ASLR is known to be by-passable, and has been investigated in the literature [25], [22].

B. Code Randomization

Code Randomization has been one of the most common defense against code re-use attacks. Randomization techniques for achieving code randomization includes instruction and basic block reordering, register re-allocation, instruction substitution and NOP insertion. These techniques are implemented by [13], [16] in the compiler. [14], [15] and [17] implements these techniques in virtual machines and JIT. [20], [24] uses static binary re-writing to implement above techniques. [15] applies above techniques on dynamically generated code. These implementations show that diversification has a positive impact on software security with negligible performance overhead. [15] relies on static NOP insertion techniques while [17] has shown that even less performance overhead could be achieved using dynamic adaptive NOP insertion. Control flow integrity (CFI)[7], [10] have also been introduced in literature to prevent memory exploits, especially return-oriented programming attacks (ROP).

VIII. Conclusion

In this paper, we presented `RandHeap`, a technique to prevent and detect heap spray attacks. `RandHeap` randomizes the heap in three ways: (i) by randomizing object layouts, (ii) by randomizing the array layout, and (iii) encrypting data on the heap. `RandHeap` randomizes the object layout for all classes by randomizing the position of fields and adding random padding between them. `RandHeap` decides a random layout for all the arrays by randomizing the element locations

indexArrayLength	Cases for inserting padding	Element Arrangements	Possible array layouts
2	2^2	2!	8
4	2^4	4!	384
8	2^8	8!	10^7
16	2^{16}	16!	1.3×10^{18}
32	2^{32}	32!	1.1×10^{45}

TABLE I: Possible Array Layouts for different values of `indexArrayLength`

and adding padding between them. `RandHeap` encrypts data to be written on the heap by `XOR`ing the data with a random key before writing it and decrypts the data before reading by `XOR`ing it by the same key.

We integrated `RandHeap` with Google V8 and JikesRVM, and evaluated `RandHeap`'s performance, security and memory requirements.`RandHeap` produces a low execution overhead of 6% when integrated with Google V8 and JikesRVM. Google V8 with `RandHeap` requires 6-9% of extra memory, while JikesRVM with `RandHeap` requires 5-10% of extra memory. We also implemented heap spray attack in current literature in both Google V8 and JikesRVM. We found that `RandHeap` could detect all attacks before the execution of malicious code commences.

REFERENCES

[1] Google v8. https://developers.google.com/v8/.

[2] Heap feng shui in javascript. https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf.

[3] Heapshield: Library-based heap overflow protection for free. https://people.cs.umass.edu/~emery/pubs/06-28.pdf.

[4] Libsafe. http://directory.fsf.org/wiki/Libsafe.

[5] Octane 2.0 javascript benchmark. http://octane-benchmark.googlecode.com/svn/latest/index.html.

[6] Time of check to time of use. https://en.wikipedia.org/wiki/Time_of_check_to_time_of_use.

[7] ABADI, M., BUDIU, M., ERLINGSSON, Ú., AND LIGATTI, J. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005* (2005), pp. 340–353.

[8] ALPERN, B., AUGART, S., BLACKBURN, S. M., BUTRICO, M., COCCHI, A., CHENG, P., DOLBY, J., FINK, S., GROVE, D., HIND, M., MCKINLEY, K. S., MERGEN, M., MOSS, J. E. B., NGO, T., SARKAR, V., AND TRAPP, M. The Jikes Research Virtual Machine project: Building an open source research community. *IBM Systems Journal 44*, 2 (May 2005).

[9] BLACKBURN, S. M., GARNER, R., HOFFMAN, C., KHAN, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (New York, NY, USA, Oct. 2006), ACM Press, pp. 169–190.

[10] BUROW, N., CARR, S. A., BRUNTHALER, S., PAYER, M., NASH, J., LARSEN, P., AND FRANZ, M. Control-flow integrity: Precision, security, and performance. *CoRR abs/1602.04056* (2016).

[11] DING, Y., WEI, T., WANG, T., LIANG, Z., AND ZOU, W. Heap taichi: exploiting memory allocation granularity in heap-spraying attacks. In *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010* (2010), pp. 327–336.

[12] GADALETA, F., YOUNAN, Y., AND JOOSEN, W. *BuBBle: A Javascript Engine Level Countermeasure against Heap-Spraying Attacks*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 1–17.

[13] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 40–40.

[14] HISER, J., NGUYEN-TUONG, A., CO, M., HALL, M., AND DAVIDSON, J. W. Ilr: Where'd my gadgets go? In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 571–585.

[15] HOMESCU, A., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. Librando: transparent code randomization for just-in-time compilers. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (2013), pp. 993–1004.

[16] HOMESCU, A., NEISIUS, S., LARSEN, P., BRUNTHALER, S., AND FRANZ, M. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (Washington, DC, USA, 2013), CGO '13, IEEE Computer Society, pp. 1–11.

[17] JANGDA, A., MISHRA, M., AND SUTTER, B. D. Adaptive just-in-time code diversification. In *Proceedings of the Second ACM Workshop on Moving Target Defense, MTD 2015, Denver, Colorado, USA, October 12, 2015* (2015), pp. 49–53.

[18] KIL, C., JUN, J., BOOKHOLT, C., XU, J., AND NING, P. Address space layout permutation (ASLP): towards fine-grained randomization of commodity software. In *22nd Annual Computer Security Applications Conference (ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA* (2006), pp. 339–348.

[19] NOVARK, G., AND BERGER, E. D. Dieharder: Securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 573–584.

[20] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Practical software diversification using in-place code randomization. In *Moving Target Defense II - Application of Game Theory and Adversarial Modeling*. 2013, pp. 175–202.

[21] RATANAWORABHAN, P., LIVSHITS, B., AND ZORN, B. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th Conference on USENIX Security Symposium* (Berkeley, CA, USA, 2009), SSYM'09, USENIX Association, pp. 169–186.

[22] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013* (2013), pp. 574–588.

[23] STOJANOVSKI, N., GUSEV, M., GLIGOROSKI, D., AND KNAPSKOG, S. J. Bypassing data execution prevention on microsoft windows XP SP2. In *Proceedings of the The Second International Conference on Availability, Reliability and Security, ARES 2007, The International Dependability Conference - Bridging Theory and Practice, April 10-13 2007, Vienna, Austria* (2007), pp. 1222–1226.

[24] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 157–168.

[25] ZHUANG, Y., ZHENG, T., AND LIN, Z. Runtime code reuse attacks: A dynamic framework bypassing fine-grained address space layout randomization. In *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013.* (2014), pp. 609–614.