On the Analysis of Byte-Granularity Heap Randomization

Daehee Jang, Jonghwan Kim, Hojoon Lee, Minjoon Park, Yunjong Jung, Minsu Kim, and Brent Byunghoon Kang, *Member, IEEE*

Abstract—Heap randomization, in general, has been a well-trodden area; however, the efficacy of byte-granularity randomization has never been fully explored as misalignment raises various concerns. Modern heap exploits often abuse the determinism in word alignment, and modern CPU architecture better supports unaligned access (since Nehalem). Based on such new developments, we conduct an in-depth analysis of evaluating the efficacy of byte-granularity heap randomization in three folds: (i) security effectiveness, (ii) performance impact, and (iii) compatibility analysis to measure deployment cost. Security discussion is based on 20 CVE case studies. To measure performance details, we conduct cycle-level microbenchmarks and report that the performance cost is highly concentrated to edge cases depending on the L1-cache line. Based on such analysis, we design and implement an allocator suited for byte-granularity heap randomization. On the negative side, our analysis suggests that byte-granularity heap randomization has high deployment cost due to various implementation conflicts. We enumerate the problematic compatibility issues using Coreutils, Nginx, and ChakraCore benchmarks.

Index Terms—Heap, exploit, unaligned access, randomization, allocator

1 Introduction

Memory corruption vulnerabilities are widely exploited as attack vectors. According to the recent statistics from the Common Vulnerabilities and Exposures (CVE) database, the majority of the arbitrary code execution exploits that have a CVSS score greater than 9.0 are caused by heap-based vulnerabilities such as use-after-free and heap overflow [32]. Recent vulnerability reports [36] also suggest that most of the usefully exploitable bugs are typically caused by heap corruptions. So far, numerous heap randomization approaches have been proposed. However, some remnants of vulnerabilities still survive due to their exploitation primitives. In modern heap exploit attacks, type confusions between integer-pointer or float-pointer are often utilized [26], [27]. To trigger such confusion, a crafted object/pointer spraying technique is required. These advanced heap exploitation techniques often take advantage of the fact that although the overall heap layout is unpredictable, the pointer-width alignment of any heap chunk is deterministic (the chunk address is always divisible by `sizeof(void*)`).

Several heap-related defense approaches, including [37], [38], [39], [51], [56], [57], [60], have provided insights into making the heap exploitation more difficult by effectively randomizing the heap layout. However, previous methods avoids reducing this randomization granularity into byte-level as it breaks the "CPU word alignment" for heap memory access. For example, [57], [68] and [39] randomizes the heap layout by prepending a random sized dummy memory space between each heap chunk; however, the randomized distance between heap chunk is guaranteed to be divisible by `sizeof(void*)` to respect the CPU word granularity memory alignment. In the case of [43], the paper suggests the importance of reducing the memory allocation granularity[1] for heap defense. However, the paper considers the pointer-width granularity (8-bytes) as the smallest possible allocation in their discussion. The main reason why previous work avoided byte-granularity randomization is that because modern hardware and software are often optimized to specific memory alignment.

Recently, major CPU vendors such as Intel and ARM recently started putting efforts to support arbitrarily misaligned (byte-granularity) memory access from hardware level [42], [46]. Based on such development, we conduct in-depth analysis on the efficacy of byte-granularity heap randomization in three folds. First, we analyze the security impact of adopting byte-granularity randomness to heap chunks. Second, we conduct cycle-level microbenchmark and memory intensive application benchmarks to reveal the performance impact of byte-granularity randomization. Finally, we analyze various compatibility problems that byte-granularity randomness can cause.

For evaluation purpose, we design and implement an allocator which adopts byte-granularity randomness to heap chunk allocation with specific exception handling and

- *D. Jang is with the Georgia Institute of Technology, Atlanta, GA 30332 USA. E-mail: daehee87@gatech.edu.*
- *J. Kim, M. Park, Y. Jung, and B.B. Kang are with the Korea Advanced Institute of Science and Technology, Yuseong-gu, Daejeon 34141, South Korea. E-mail: {zzoru, dinggul, yunjong, brentkang}@kaist.ac.kr.*
- *H. Lee is with the Sungkyunkwan University, Jongno-gu, Seoul 03063, South Korea. E-mail: hojoon.lee@skku.edu.*
- *M. Kim is with the National Security Research Institute, Yuseong-gu, Daejeon 34044, South Korea. E-mail: pshskms@nsr.re.kr.*

Manuscript received 4 Oct. 2018; revised 7 Mar. 2019; accepted 1 Oct. 2019. Date of publication 16 Oct. 2019; date of current version 1 Sept. 2021. (Corresponding author: Daehee Jang.)
Digital Object Identifier no. 10.1109/TDSC.2019.2947913

1. To quote the paper: "memory managers at all levels should use finer memory allocation granularity for better security"

optimization. We name our allocator "Randomly Unaligned Memory Allocator (ra-malloc)". Performance regarding chunk management and allocation speed of ra-malloc does not outperform traditional heap allocators. However, ra-malloc leverages architecture specifics and reduces memory access penalty of byte-granularity heap randomization.

To measure the performance impact of ra-malloc, we apply various allocators (ra-malloc, tcmalloc, dlmalloc, jemalloc and ottomalloc) to SPEC2006 and compare their benchmark results. To investigate compatibility problem of byte-granularity heap randomization, we use Coreutils utilities test suite, Nginx web server test suite, and ChakraCore JavaScript engine test suite. We discuss the various issues regarding compatibility in Section 5 and summarize negative analysis results as limitations in Section 6.

The contributions of this paper include the followings:

- This is the first work that fully analyzed the pros and cons of byte-granularity heap randomization.
- We conducted in-depth experiments and security analysis caused by byte-granularity randomness.
- We design and implement a memory allocator (ra-malloc) that optimizes the byte-granularity heap randomization on recent Intel architecture.

2 Security Analysis of Byte-Granularity Heap Randomization

In modern heap exploitation, achieving direct manipulation of code pointer is unlikely due to the state-of-the-art compiler defenses such as VTGuard, pointer encryption, and VT-read-only checks [62]. As a result, the surviving remnants of heap exploitation techniques often target data pointer and involve convoluted steps of pointer manipulation to complete the exploit. The key procedure is controlling the program to use the attacker-crafted non-pointer values as an intact pointer. Exploitation of typical heap vulnerabilities such as use-after-free, out-of-bounds access (including heap overflow), type confusion and uninitialized access mostly involves multiple steps of *invalid pointer dereference* to trigger information leakage, achieve arbitrary memory access and finally, execute arbitrary code.

Traditional heap spray places a huge amount of NOP-sled and shellcodes into predictable heap address[2] as direct code pointer manipulation was easy, however, the goal of modern heap spray is more focused on placing crafted pointers around out-of-bounds heap area. Because heap spray allows an attacker to control a broad range of heap region, malicious pointer-spraying could be a threat without any (for 32-bit address space) or with only limited (for 64-bit address space) information disclosure. In this section, we first clarify the attack model and assumptions, then discuss the security effectiveness of byte-granularity heap randomization in three terms: (i) successful triggering of heap vulnerability (ii) information leakage attacks (iii) bypassing byte-granularity heap randomization.

2.1 Successful Triggering of Heap Vulnerabilities

Any triggering step of heap vulnerabilities that occurs due to *out-of-bounds* access[3] are affected by byte-granularity heap randomization. For example, the first use of dangling-pointer in use-after-free guarantees to crash any application with 87.5 percent (75 percent in 32-bit) probability as there are eight (four in 32-bit) possible outcomes of the misinterpreted pointer alignment.

Consider the exploitation steps of use-after-free: (i) an object is *freed* and a dangling pointer is created, (ii) the attacker places a crafted object around the dangling-pointed memory region, and (iii) the program *uses* the dangling pointer as if the original object member variables (pointer member variables) are still intact thus using attacker's crafted pointer. These steps imply that there are two independent heap chunk allocations around the dangling-pointed heap area. Although the address of each heap chunks is random, if the allocation granularity is bigger than the pointer-width, an attacker can spray the heap and overlap the fake object and dangling-pointer thus successfully trigger the use-after-free without pinpointing the exact memory addresses.

This effectiveness can be described by depicting a simplified example. Fig. 1 depicts an example case of dereferencing a dangling pointer (to access a pointer member variable) after attacker launches a pointer-spray attack. For simplicity, let's assume attacker wants to hijack a pointer into `0xdeadbeefcafebabe` and there are five unpredictable cases of dangling pointers which will be randomly decided at runtime.

In Fig. 1a, an attacker can hijack the target pointer member variable with a very high chance because the heap randomization follows word-granularity. The attacker can spray the eight-byte sequence "`DE AD BE EF CA FE BA BE`" sufficiently long to defragment the heap region and bypass the randomization. However in Fig. 1b, the randomization is byte-granularity thus the attack fails with 87.5 percent probability regardless of the spray; unless the pointer is composed with same bytes (we discuss this issue at the end of this section).

The same effectiveness can also be observed from heap overflow situation. In general, randomizing the distance between the source of the out-of-bounds access (buggy buffer) and the adjacent heap object is an effective mitigation approach against heap overflow attack. However, if the heap allocation granularity is bigger than the width of the pointer, the effectiveness of this approach can be reduced regarding pointer manipulation. This can be demonstrated with a hypothetical situation in which the attacker tries to hijack a pointer value inside an adjacent heap object.

The effectiveness of byte-granularity heap randomization is not specific to particular heap vulnerabilities. We emphasize that *any exploitation step which involves the use of crafted pointer upon out-of-bounds heap access is affected*. For example, exploitation of heap overflow, uninitialized heap access vulnerability also involves out-of-bounds heap access [8], [13] thus affected by byte-granularity heap randomization.

2. Heap address prediction via massive heap spray is feasible under 32-bit address space layout randomization. In case of 64-bit address space layout randomization, heap spray becomes equally feasible if a heap segment base address is exposed.

3. In this paper, out-of-bounds access indicates memory access that crosses heap chunk bound.

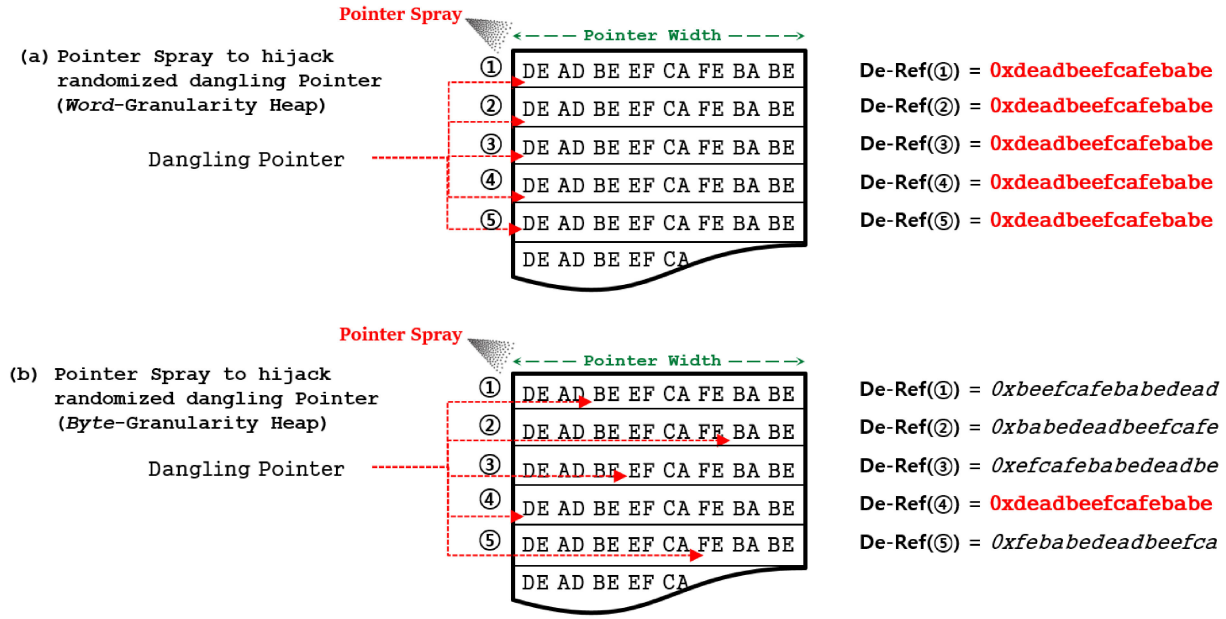


Fig. 1. Simplified example cases of dereferencing a dangling pointer for Use-After-Free under pointer-spray attack (assuming `0xdeadbeefcafebabe` is the crafted pointer). The box in the middle represents dangling pointed object and each row indicates pointer-type member variable. Assume there are five possible dangling pointers due to randomization. For better visualization, the memory dump is shown in big-endian format.

So far, the security effectiveness of byte-granularity heap randomization seems small, as one out of eight (or four) triggering attempts will succeed. However, this probability of single dereferencing is not the probability of a successful attack. Modern heap exploitation usually involves multiple combination and repetition of such bug triggering. According to Google Project-Zero, successful exploitation of CVE-2015-3077 required up to 31 times of pointer confusion. As heap exploitation involves multiple uses of crafted pointers, the defense probability will increase exponentially. However exact calculation of defense probability based on a number of the crafted pointer is unrealistic as modern heap exploitation usually achieves complete information leakage capability in the middle of the exploitation. In next subsection, we discuss the effectiveness of byte granularity heap randomization considering information leakages.

2.2 Information Leakage

Information leakage vulnerabilities are typically discovered in software that implements ECMAScript [29] parsers including ChakraCore, V8, ActionScript, and Spider Monkey. It is well established that *complete disclosure of memory* renders all the randomness-based exploit defenses ineffective. In this subsection, we analyze the impact of byte-granularity heap randomization over various types of information leakage attempts.

According to previous insights of past research, there are information disclosure attacks based on side-channels which do not leverage memory corruption [30], [41], [45], [52], [54], [63], [64], [67]. Memory randomization does not affect such attacks. However, some information disclosure attacks are based on memory corruption. In such case, byte-granularity randomization could affect information leakage as well. We analyzed the impact of byte-granularity randomization for such cases.

One of the information leakage primitive found in real-world CVEs is often achieved by OOB (Out-Of-Bound read/write) vulnerability. Such OOBs can be divided into several types: (i) direct disclosure of arbitrary memory contents without involving use of fake pointer, (ii) partial disclosure of non-arbitrary memory, and (iii) indirect disclosure of memory contents. Example case of (i) would be literal array (e.g., string) based OOB [3], [10]. If the OOB is achieved by such size-corrupted array (e.g., string with negative size) and the memory content of size-corrupted array is directly readable by the attacker, byte-granularity heap randomization has no benefit over existing coarse-grained heap randomization. Attacker can read any heap region starting from the OOB heap by giving arbitrary substring index and length (e.g., `leakedarray.substr(0x7fff4000 - [leakedarray address], 100)` to read 100 bytes from arbitrary address `0x7fff4000`).

On the other hand, in the case of (ii) and (iii), heap allocation granularity can affect the information disclosure attempt. For example, size-shrink based OOB [12] allows an attacker to overread out-of-bound heap buffer. In such case, attacker can read memory contents adjacent to the end of heap buffer. In order to achieve complete memory access capability, an attacker hijacks an object that has backstore pointer.[4] Due to heap isolation, spraying arbitrary object at arbitrary heap segment is not straightforward. Fig. 2 illustrates such an example case. From Fig. 2, information leak requires fake pointer spraying in case attacker cannot place a backstore pointer (`ptrB` in the figure) inside directly accessed out-of-bound heap region.

In case the OOB is based on object array, the attacker cannot directly read the memory contents because the type of array element is not a literal but an object. ECMAScript

4. A leaf pointer that ultimately being used for reading/writing memory. (e.g., "DataView" in Chrome V8).

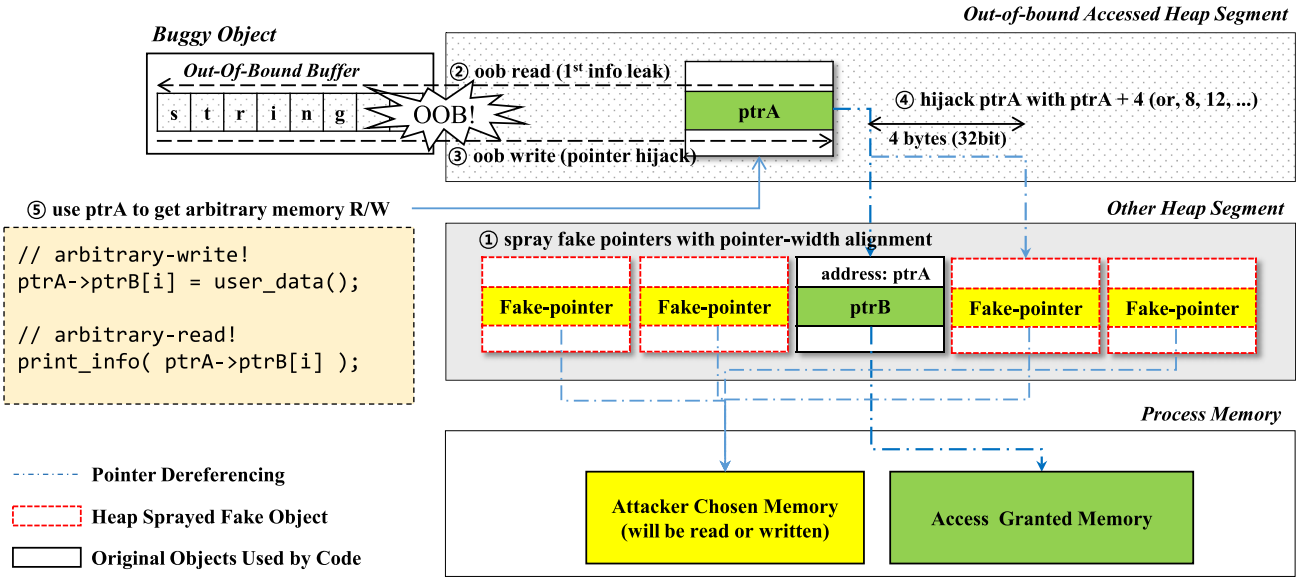


Fig. 2. Achieving arbitrary read/write primitive from partial out-of-bound read/write vulnerability. If the backstore pointer (ptrB in this case) is inside the out-of-bounds heap area, byte-granularity heap randomization can be bypassed. However, if the backstore pointer and out-of-bound heap is at different heap segments, fake pointer spraying is required.

parsers do not provide any syntax that dumps the raw memory contents of an object. Therefore, an additional trick is required to turn such OOB into working information leakage. Based on our analysis, the attacker can spray fake objects around out-of-bound heap area and making the application to use the fake object. Use of such fake object triggers further type confusions and enables the attacker to hijack a backstore pointer. Crafting out-of-bound heap memory layout for proper pointer confusion often requires predictable memory alignment. Fig. 3 illustrates such case (CVE-2016-0191) where the information leakage is achieved via object spraying technique. In this exploitation, attacker achieves information leakage primitive by making the ChakraCore confuse the reference of the pointer inside `JavaScriptDate` object as the pointer of a `DataView` object. The exploitation repeats spraying the fake `DataView` object and using the dangling pointer of `JavaScriptDate` object until two pointers are confused.

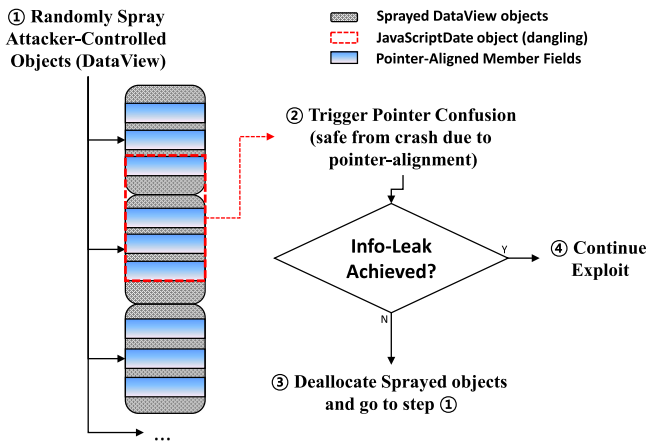


Fig. 3. Information leakage steps of CVE-2016-0191 against Edge. Attack is dependent to heap randomization of pointer-width allocation granularity.

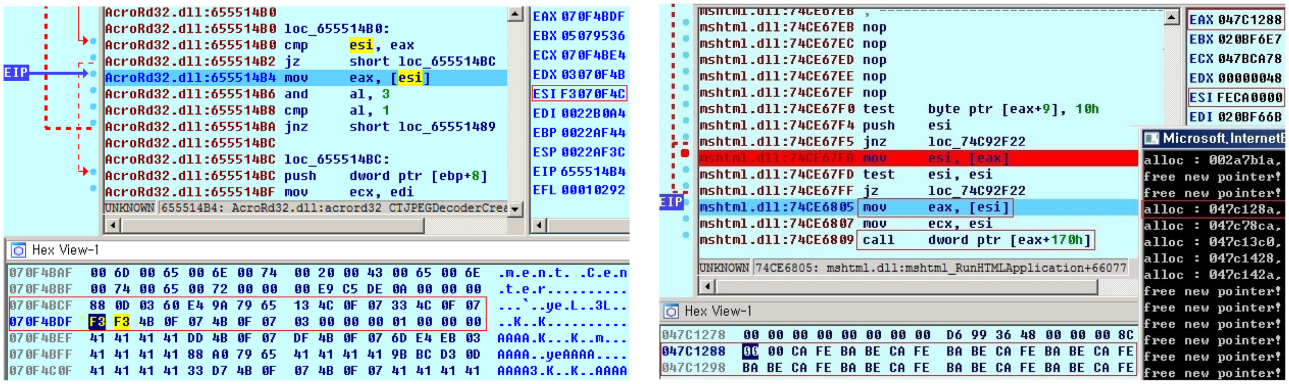
In the case of CVE-2016-1677 [19], class type confusion vulnerability enables an attacker to read lower 16 bits of a pointer value inside a particular object. This leaked information (lower 16 bits of a pointer) can be used to significantly reduce the entropy of ASLR for the memory segment pointed by the partially leaked pointer.

In the case of CVE-2016-1665 [18], a miscalculated index value for an object array (`Node` array in V8) allows out-of-bound object dereferencing that is outside the array. In order to get information leakage with this bug, sophisticated heap spraying is required to place a fake pointer adjacent to this array, which allows him/her to construct his/her own `Node` object. Later, the program references a short (2-byte word) pointer inside the attacker's fake object to decide a branch path. In such a case, the branch path of the program is decided by a fake value that is dereferenced by a fake pointer. Similarly, in [20], an attacker is allowed to decide the program control flow depending on the 1-byte data read from the out-of-bounds heap. In [18] and [20], a side channel attack can be applied in order to reveal the referenced bytes as the control flow is affected by such values.

To summarize the analysis, byte-granularity heap randomization does not affect information leakage which directly reveals arbitrary memory contents without involving fake pointers. However, if the information leakage is (i) dependent on illegal use of fake pointers, or (ii) partial, byte-granularity heap randomization hinders the attack.

2.3 Bypassing Byte-Granularity Randomization

Byte granularity heap randomization guarantees four (or eight) possible cases against any pointer manipulation due to out-of-bounds access. In turn, an attacker who wishes to hijack a single pointer (or any word-unit data), say, with a value of `0x1122334455667788`, stands a 87.5 percent chance of failing in his/her overwrite attempt with pointer spraying (i.e., `0x8811223344556677`, `0x7788112233445566`, etc). Thus, a plausible way of



(a) CVE-2013-2729 (b) CVE-2013-0025

Fig. 4. Reproducing CVE-2013-2729 (Acrobat Reader) and CVE-2013-0025 (Internet Explorer) exploitation for case study.

bypassing byte granularity randomization is by constructing the entire chain of the exploit payload with *byte-shift-independent values only*. A byte-shift-independent value is a word (or doubleword) composed of the same bytes (e.g., `0x9797979797979797` in a 64-bit system or `0x35353535` in a 32-bit system).

At this point, the byte-shift-independent values are always invalid virtual addresses in current 64-bit system. In 32-bit system, such values can be predictable and valid address especially if the attacker allocates a sufficiently large region of the memory (i.e., allocating the lower 1-GByte of memory in a 32-bit address space will include an address of `0x35353535`). It could be a threat to byte-granularity heap randomization if attacker can place the crafted chunk at such address and construct the crafted pointers with byte-shift-independent values. Virtual addresses such as `0x35343534` make the same effect with 50 percent probability. To address this issue, 32-bit version of ra-malloc provides configuration for avoiding such addresses similarly as EMET [44]. The difference between EMET and 32-bit ra-malloc is that while EMET initially pre-allocate (mmap) blacklist pages, 32-bit ra-malloc checks the address at allocation time and re-allocates the chunk. We compiled 32-bit version of SPEC2006 suites and tested performance impact of this algorithm. The algorithm caused up to 3 percent overhead in 32-bit SPEC2006 benchmark. Detailed algorithm description for avoiding such addresses is described in Section 4.

3 Case Studies

Quantifying the probability of successful heap vulnerability exploitation is an arduous process and furthermore, proving the correctness of such probability (of successful exploitation) is practically impossible. The claim we can make is that the byte-granularity heap randomization foils *the inter-chunk use of fake pointers* with 75 Percnt in a 32-bit system and 87.5 Percnt in a 64-bit system. Here we discuss the necessity of fake pointer spraying and pointer-width allocation granularity in real-world heap exploit cases.

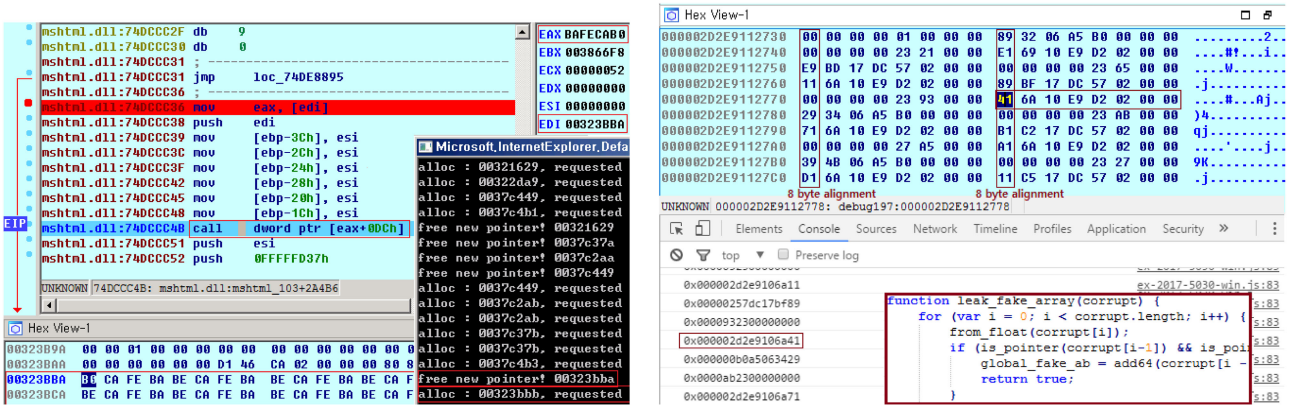
To analyze the security effectiveness of byte-granularity heap randomization, we investigate publicly disclosed heap memory corruption vulnerabilities [1], [2], [4], [5], [6], [7], [8], [11], [13], [14], [15], [16], [17], [21], [22], [23], [24], [25], [26], [27] that enabled attackers to achieve information leakage or arbitrary-code-execution against various software mostly from Pwn2Own contest and Google bug bounty program.

Throughout case studies, four of the attacks [1], [2], [5], [27] were reproduced and byte-granularity heap randomization was partially applied[5]. Rest of the case studies are conducted based on debugging minimal proof-of-concept codes with documented information. We describe details of reproduced cases then summarize the overall results.

CVE-2013-2729 is a heap vulnerability that existed in Acrobat Reader X. An integer overflow vulnerability occurs while parsing a bitmap image embedded in a PDF document, which eventually leads to out-of-bound heap access. As the full exploit code is available on the Internet, we reproduced the entire attack under debugging environment to see how byte granularity heap randomization hinders the exploitation. The PoC reliably achieves arbitrary code execution against 32-bit Acrobat Reader X in Windows 7; however, after we adopt byte-granularity randomness for heap allocation[6], the PoC exploit constantly crashes due to dereferencing the invalid pointer (incorrectly aligned due to byte granularity randomization). However, the program never crashed while processing benign PDF documents. For example, the debug screen Fig. 4a shows an example crash due to out-of-bound heap pointer access (ESI is holding an invalid fake pointer). The value of ESI is `0xF3070F4C`, which is a 1-byte shifted value from the intended one (`0x070F4C33`). This occurs because of the unpredictable change in the pointer-width alignment between the out-of-bound buffer and the target heap chunk. Although this single step of pointer hijacking could be successfully operated in 25 Percnt of the cases, the success probability of first information leaking step (leaking a heap pointer) was less than 1 Percnt due to multiple needs of out-of-bound fake pointer access.

5. Adopting byte-granularity heap randomization to legacy binary is not fully applicable except old version of Acrobat Reader due to compatibility issues which will be discussed later. In such case, we applied byte-granularity randomization only against exploit-relevant objects by identifying the object based on its size and allocation site for analysis. Details of such issues will be discussed in Section 4

6. by hooking the heap allocation APIs in the AcroRd32.dll import address table



(a) CVE-2012-4792 (b) CVE-2017-5030

Fig. 5. Reproducing CVE-2012-4792 (Internet Explorer) and CVE-2017-5030 (Chrome) exploitation for case study.

CVE-2013-0025 is a use-after-free vulnerability that existed in Internet Explorer 8. We reproduced the triggering of use-after-free with proof-of-concept code and debugged the vulnerability. With this vulnerability, a `CParaElement` object is used by referencing a `CDoc` object after it has been freed. The dangling pointer of `CDoc` dereferences the pointer of `CParaElement`; then, the pointer of `CParaElement` dereferences the pointer of the C++ virtual table. Finally, the pointer of the virtual table dereferences a function pointer. From the debugging screen of Fig. 4b, there are three successive pointer dereferencing before EIP is changed (e.g., `call dword ptr [eax+170h]`). The PoC exploit failed the first step of pointer hijacking "`mov esi, [eax],`" which was supposedly aimed at hijacking the pointer of `CParaElement` into `0xBEBAFECA` (we chose this value for demonstration), which is a sprayed fake pointer value to point fake `CParaElement` object.

At the bottom right of the debug screen of Fig. 4b, there is an allocation log that reports that a heap chunk was yielded at `0x047c128a` (2-byte broken alignment). However, the value of the dangling pointer is `0x047c1288` (0-byte broken alignment). Because the randomly broken alignment of the dangling pointer differs from that of the recycled pointer, the hijacked value of `ESI` becomes `0xFECA0000` (the two zero bytes are remnants from the original object), which was not the intended value. To hijack the EIP with this use-after-free, more fake pointer spraying attacks are required. This vulnerability does not give the attacker any information leakage. Therefore arbitrary code execution can be achieved under two circumstances: (i) attacker can brute-force the address with extensive heap spray in 32-bit address space, (ii) attacker achieves information leak via other independent vulnerability. If the information leakage is enough to disclose entire memory, the randomness is useless, however, if the address prediction is based on extensive heap spray or partial information leakage (leaking few pointers) byte granularity heap randomization becomes effective to hinder pointer hijacking.

CVE-2017-5030 is a V8 out-of-bound access vulnerability which gives information leakage. In this vulnerability, an attacker can achieve literal based information leakage, thus reveal the out-of-bound accessed heap memory contents directly. Using this initial information leakage, the attacker gets useful information which will be later used. Partial information leakage itself cannot achieve arbitrary code execution. Thus attacker triggers heap spraying and pointer confusion techniques to take control over a backstore pointer. Using the hijacked backstore pointer, the attacker places shellcode inside JIT memory segment (which is RWX) and triggers the code execution. Initial information leakage in this exploit does not involve out-of-bound fake pointer access, and the memory content is exposed directly as literals (float type array). Therefore byte-granularity heap randomization has no mitigation effect against this vulnerability.

The memory scanning steps (analyzing the leaked memory contents) in this exploit involves pointer identification technique based on 8-byte alignment. The essence of this technique is looking up the least significant bit of each 8-byte chunks assuming they are word-aligned. If the least significant bit of leaked word is set, attacker assumes it as a pointer (V8 stores pointer in this manner). Fig. 5b Shows memory dump of V8 and the leaked memory contents from Chrome debugging console while the exploitation is in progress. From the figure, attacker assumes the leaked value `0x000002d2e9106a41` (highlighted with a box) is a pointer based on the observation that least significant byte among the 8-byte is an odd number. The adoption of byte-granularity heap randomization breaks this information leak analysis as an attacker can no longer assume the 8-byte pointer alignment of out-of-bound heap region. Under the byte-granularity heap randomization, information leakage analysis requires concrete sentinel such as attacker crafted length value.

CVE-2012-4792 is a use-after-free vulnerability that existed in Internet Explorer 8. The vulnerability is caused by reusing a dangling pointer of `CButton` object after it has been freed. The button object referenced by a dangling pointer in CVE-2012-4792 has a size of `0x86` bytes, and the use-after-free logic dereferences this dangling pointer to retrieve VPTR inside the object. The proof of concept (PoC) exploit code hijacks the VPTR pointer inside `EAX` to `0xBEBAFECA` (the little-endian representation of byte stream `CA FE BA BE`). The exploit code for 32-bit environment works reliably under randomized heap (under ASLR) due to pointer spraying. However, after byte-granularity heap randomization is applied, 75 Percnt of exploit attempt fails to hijack the VPTR as intended. Fig. 5a shows the debugging screen and memory allocation log trace of Internet Explorer 8 while the PoC of CVE-2012-4792 is being triggered. From the figure, the allocation request for the attacker's data

TABLE 1
Summarized Result of Case Study

#	CVE #	Bug Description	Attack Target	Remarks
1	CVE-2013-2729	out-of-bound	Acrobat Pro (32bit)	Pointer Spray
2	CVE-2015-2411	use-before-init	IE11 (32bit)	Pointer Spray
3	CVE-2016-0191	use-after-free	Edge (64bit)	Pointer Spray
4	CVE-2016-1653	JIT compiler bug	Chrome (32bit)	Pointer Spray
5	CVE-2016-1857	use-after-free	Safari (32bit)	Pointer Spray
6	CVE-2016-5129	out-of-bound	Chrome (32bit)	Pointer Spray
7	CVE-2012-4792	use-after-free	IE8 (32bit)	Multiple Pointer Corruption
8	CVE-2013-0025	use-after-free	IE8 (32bit)	Multiple Pointer Corruption
9	CVE-2014-3176	out-of-bound	Chrome (32bit)	Multiple Pointer Corruption
10	CVE-2016-0175	use-after-free	Windows Kernel	Multiple Pointer Corruption
11	CVE-2016-0196	use-after-free	Windows Kernel	Multiple Pointer Corruption
12	CVE-2016-1017	use-after-free	Flash Player	Multiple Pointer Corruption
13	CVE-2017-5030	out-of-bound	Chrome (64bit)	Information Leak Analysis
14	CVE-2015-1234	heap overflow	Chrome (32bit)	Non-Pointer Corruption
15	CVE-2017-2521	out-of-bound	Safari	Non-Pointer Corruption
16	CVE-2016-1859	use-after-free	Safari	Non-Pointer Corruption
17	CVE-2013-0912	type confusion	Chrome (32bit)	In-bound Corruption
18	CVE-2017-0071	type confusion	Edge (64bit)	In-bound Corruption
19	CVE-2016-1016	use-after-free	Flash Player	None
20	CVE-2016-1796	heap overflow	OSX Kernel	None

yields memory address `0x00323bbb` (3-byte distance from pointer-width alignment), which has a different alignment to that of the dangling pointer `0x00323bba` (2-byte distance from pointer-width alignment). Because of the discrepancy among these two memory address, the hijacked pointer from "`mov eax, [edi]`" instruction becomes `0xBAFECAB0`. The lower byte `0xB0` is the remnant from the original chunk which is not the value controlled by the attacker.

According to our analysis, 32-bit applications often bypassed ASLR by spraying extensive amount of fake pointers [1], [2], [5], [8], [17], [22], [24]. In such cases, byte-granularity heap randomization had high impact for neutralizing the successful exploitation. However, heap corruption based on class type confusion vulnerabilities [4], [25] were not affected by byte-level allocation granularity because such errors are caused by two different interpretation against the same original object. For example, assuming there is an object with member `A` (pointer type) and `B` (integer type), type confusion error in our evaluation makes the program to think object member `B` as pointer. Therefore the fake pointer, in this case, is allocated with intact pointer `A` at the same time; thus shares *equally changed memory alignment*.

Cases such as [7], [25], [26], [27] achieved information leakage primitive without using fake pointers. Similar cases such as [6], [13], [15], [16], [23] also achieved information leakage primitive however it required multiple use of fake pointers during the process. Finally, in case of local-privilege-escalation (LPE) attacks [11], [14], [21], the efficacy of byte-granularity heap randomization was lower than we anticipated as exploitation steps involving the use of fake pointers were relatively shorter than other cases. Table 1 summarizes overall analysis results.

4 Performance Analysis of Byte-Granularity Heap Randomization

As byte-granularity heap randomization inevitably involves CPU-level unaligned or misaligned memory access, we start this section by discussing some backgrounds regarding the unaligned access and analyze our findings in the performance and compatibility such as the atomicity and the alignment fault issues involved in the misaligned access. Based on microbenchmark analysis, we design a new memory allocator ra-malloc for efficient byte-granularity randomization. We further show in-depth testing results on ra-malloc using various benchmarks.

4.1 Unaligned Memory Access

The term *alignment* can be used with many units such as page alignment, object alignment, and word alignment. In this paper, the term alignment specifically refers to the *CPU word granularity alignment*. Unaligned memory access can be observed in special cases in an Intel-based system (e.g., `#pragma pack(1)` in the C language, x86 GLIBC I/O buffer handling). However, memory accesses are always encouraged to be aligned at multiples of the CPU word size. The main reason stems from the limited addressing granularity of the hardware. In general, CPU architectures feature a memory bus line with 8-byte addressing granularity; therefore, the retrieval of memory at an unaligned address requires exceptional handling from the hardware viewpoint. Handling such unaligned memory access may involve various performance penalty issues such as a possible delay of store-load forwarding, cache bank dependency, and cache miss. The major penalty induced by unaligned memory access is closely related to the increased number of L1 cache misses. Because the CPU fetches memory contents from a cache with *cache line granularity*, unaligned memory access that crosses the boundary of two separate cache lines causes both cache lines to be accessed to retrieve a single word, causing a performance penalty.

Other than performance, unaligned memory access also raises concerns regarding memory access atomicity. Atomic memory access is a critical issue in concurrent software development. Multithreaded applications utilize various

TABLE 2
Default Configuration of CR0.AM Bit and EFLAGS.AC Bit in Major Operating Systems

OS	CR0.AM	EFLAGS.AC
Windows	disabled	disabled
Linux	enabled	disabled
OSX	disabled	disabled

Unless two bits are enabled at the same time, unaligned access does not raise an alignment fault under the Intel architecture.

types of synchronization primitives such as mutex and semaphore. The key primitive of critical section (protected by the lock) is that the execution should be atomic from the perspective of each thread. Unaligned access raises concerns when the application uses thread synchronization, lock-free algorithms, or lock-free data structures, which relies on instruction-level atomicity such as the `InterlockedCompareExchange()` function. Because single unaligned memory access can split into multiple accesses at the hardware level, the atomicity of memory access may not be guaranteed.

In fact, ARM architecture does not support such atomicity for unaligned access. Even recently, compare-and-swap (CAS) instructions in ARM (e.g., `ldrex, strex`) fail to operate if the target memory operand is unaligned. This is the major reason we conclude byte-granularity heap randomization is yet infeasible in ARM architecture. However, Intel microarchitecture (since P6) supports atomicity for such instructions even if the memory address is arbitrarily misaligned [55]. For example, CAS instructions of Intel ISA (e.g., `lock cmpxchg`) maintains memory access atomicity in all cases. The Intel official manual states that atomicity of `lock` is supported to arbitrarily misaligned memory.

Another important issue regarding unaligned memory access is the *alignment fault*. An *alignment fault* (which raises SIGBUS signal in Linux) occurs in the event of unaligned data memory access, depending on the CPU configuration. There are two configuration flags regarding the alignment fault in the Intel architecture, namely the `AM` bit in the system `CR0` register and the `AC` bit in the `EFLAGS` register. Unless such bits are enabled together, Intel architecture does not raise an alignment fault. Table 2 summarizes the default configuration of these registers in well-known operating systems.

However, there is an exception that raises an alignment fault regardless of such configuration. In the case of the Intel SSE [35] instruction set, an instruction such as `MOVNTQDA` raises an alignment fault if the target memory operand address is misaligned at a *16-byte boundary*. Normally, an arbitrary heap memory address has no guarantee to be 16 byte aligned in general. Therefore, codes that use SSE instruction usually do not assume that the given memory address for SSE operand will be 16-byte aligned. The preprocessing codes of SSE instruction checks the address alignment and properly handles the 128-bit misaligned memory portion. However, some of the compiler optimization cases (e.g., Clang -O3) predicts the chunk alignment and raises problem. If such cases are rare, one solution could be ra-malloc installs `alignment fault handler` and catches such exception to emulate the operation. This emulation method is observed in ARM Linux kernel to handle unaligned access. We discuss this limitation with other details regarding the compatibility conflicts.

4.2 Microbenchmark for Unaligned Access

Unaligned access is often observed under Intel-based system. In recent Intel microarchitectures, the performance penalty of unaligned access is being reduced since Nehalem [65]. Here, we show microbenchmark results regarding unaligned access in recent Intel CPUs. All the per-instruction benchmarks are composed of assembly code only, thus avoiding any compiler dependency. We used ten different Intel x86-64 CPUs and measured the execution time of 134,217,728 (0x8000000) iterations for memory access instructions.

Performance measurement for instruction is dependent on CPU pipeline state. To make the worst-case CPU pipeline, we used the same instruction for 48 consecutive times and repeated such 48 consecutive execution using a loop. The loop was composed of `dec ecx` and `jnz`, both of which have lower instruction latency and reciprocal throughput compared to the memory access instructions. This configuration makes the worst-case CPU pipeline for memory access. Fig. 6 shows the overall results. In the benchmark, only the register and memory were used as target operands (the immediates were not used since the instruction latency was lower).

Throughout the evaluation we have discovered that the performance penalty of unaligned access is severely biased by rare cases, Fig. 6 is one of the experiment results. In particular, the performance penalty of unaligned access is 0 percent if the accesses have entirely occurred inside the cache line. In case the unaligned access broke the L1 cache line, the performance penalty raised up to 30–70 percent. In case the unaligned access broke the border of two 4-KByte pages, the performance penalty was suspiciously high (marked as red in the figure). In the case of REP-based instructions (REP counter value is 256), the performance penalty of unaligned access was mostly 20–30 percent. To investigate the reason for the exceptionally high-performance penalty of unaligned access that crosses page border, we further used the PERF [34] benchmark and found that the dominant factor is the increased cache miss. Table 3 summarizes the PERF benchmark conducted on the Intel i7-6700 Ubuntu14.04 Server in a 64-bit environment.

Thus far, the microbenchmark in Fig. 6, suggests the performance penalty of unaligned access in modern Intel architecture is high only when the access crosses the border of two cache lines, and extremely high if the access crosses two-page boundaries.

4.3 Ra-Malloc

According to the cycle-level instruction benchmark, unaligned access in L1 cache line border and page border hindered the performance. To avoid such memory access while enabling byte-granularity heap randomization, we implemented a byte-granularity randomized heap allocator suited for Intel architecture. The goal of this allocator is to randomize the heap chunk location with *byte granularity* while minimizing the unaligned memory access occurs at the border of L1 cache line and page boundary. We implemented ra-malloc as part of

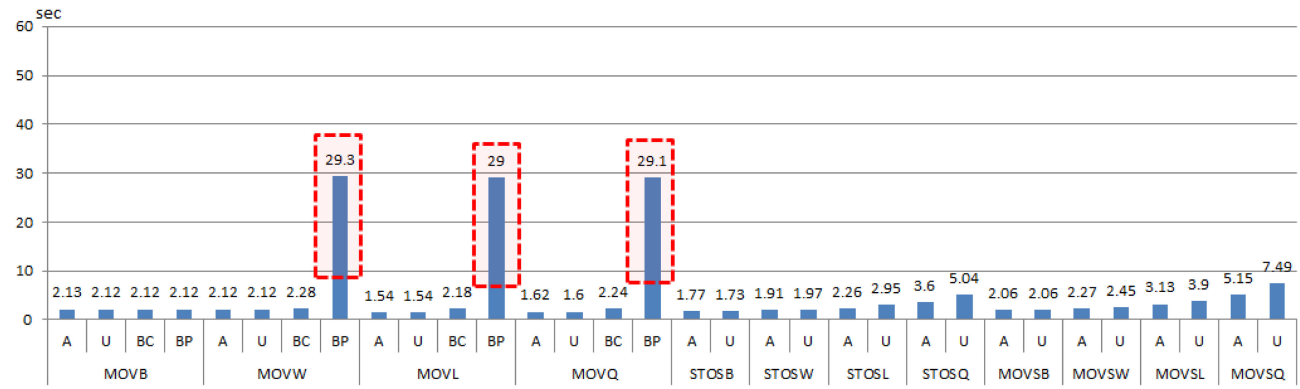


Fig. 6. Instruction-level microbenchmark result for Intel i7-6500 (SkyLake). The Y-axis represents the time (consumed seconds) for repeatedly executing the unrolled instructions 134,217,728 (0x8000000) times. `A` stands for Aligned access, `U` stands for Unaligned access inside cache line, `BC` stands for unaligned access on Border of (L1) Cache line, `BP` stands for unaligned access on Border of Pages. We ran the same benchmark for additional 9 Intel CPUs (Core-i5, Xeon, and so forth) and verified similar results in all cases.

Clang runtime, and wrote an LLVM pass for automated allocator substitution.

To break the pointer-width allocation granularity, ra-malloc additionally allocates `sizeof(void*)` additional memory space in addition to original allocation request size to reserve `sizeof(void*)` dummy space. After the allocation algorithm selects the proper location for the new chunk, ra-malloc yields the address which is randomly increased in bytes between zero and `sizeof(void*)-1`. We note that the front-end implementation (adding random numbers) for applying byte-granularity randomization is a simple task and not our main contribution. The main contribution of ra-malloc is minimizing the impact of byte-granularity randomness based on the previously discussed analysis and experiments.

The backend allocation algorithm of ra-malloc is mainly based on `jemalloc`, where the heap space is organized as multiple pools each holding objects of certain size class. However, ra-malloc considers two special cases: (i) chunk size less than L1 cache line width, (ii) chunk size bigger than L1 cache line less than page size. The size of L1 cache line (usually 64-byte or 128-byte) is dynamically calculated during the allocator initialization and page size is statically assumed to be 4 KB. Once the allocation size is determined, ra-malloc searches for an available chunk based on `jemalloc` algorithm with additional constraints that minimize the cases where chunks are spanning across special memory borders.

For object allocation smaller than L1 cache line size (including the additional space for randomization), ra-malloc guarantees the memory location of the chunk to fit between two L1 cache line borders thus eliminate any performance penalty due to byte-granularity heap randomization. In case the requested size is bigger than L1 cache line and yet smaller than page, ra-malloc places the chunk between page boundaries therefore avoid page boundary access. If the allocation size is bigger than a page, there is no additional handling as baseline allocator guarantees minimal border access without any additional handling.

TABLE 3
PERF Benchmark Revealed the Reason of Exceptionally High Performance Penalty of Unaligned Memory Access that Crosses the Border of two Pages, which is the Increased Cache Miss

Configuration	# of Page-Faults	# of Cache-Miss
Fully Aligned	45	7,647
Breaking Cache Line	46	7,527
Breaking Page Border	45	76,181

In case the application is 32-bit, ra-malloc uses address filtering algorithm to handle the case of byte-shift-independant-pointers Section 2 which could bypass ra-malloc. In case the requested chunk size is larger than `0x01010101` bytes, it is impossible to remove the byte-shift-independent address from the virtual address mapping. However, in case the size is smaller than `0x01010101` bytes, 32-bit ra-malloc ensures there would be no byte-shift-independent address inside the allocated chunk. The address checking algorithm is executed after the chunk selection and right before delivering the chunk to the application. If the chunk includes 32-bit byte-shift-independent address (e.g., `0x11111111`), ra-malloc keeps the chunk internally and allocate a new chunk. The algorithm for address inspection is as follow: (i) Calculate most-significant-byte (MSB) of chunk start address. For example, if the chunk address is `0x12345678`, MSB is `0x12`. (ii) Check if MSB-only address (e.g., `0x12121212`) is in between chunk start and end. (iii) Increase MSB by one and repeat step (i), and (ii).

Overall, the efficacy of ra-malloc would be optimal when all objects are small (less than L1 cache-line would be ideal). In reality, however, there are large heap objects. As the size of an object becomes bigger, the chance of having a costly unaligned access will increase even though ra-malloc minimizes the occurrence of unaligned access. To investigate the chunk size distribution in common applications in general, we traced heap allocation and deallocation requests of Acrobat Reader and Internet Explorer. Fig. 7 shows the object size distribution of live chunks when Acrobat Reader and Internet Explorer are running. The results are based on allocation/deallocation/reallocation call trace. The left side graph in the figure is the result of Internet Explorer after rendering the Google index page, and the right side graph is the result of Adobe Reader after rendering an ordinary PDF document.

Fig. 7 suggests that average heap chunk sizes are usually small. Byte granularity heap randomization using ra-malloc can fully avoid unaligned access penalty if all chunks are smaller than L1 cache line length. In case the chunk is larger than L1 cache line, it is inevitable to place a chunk across two

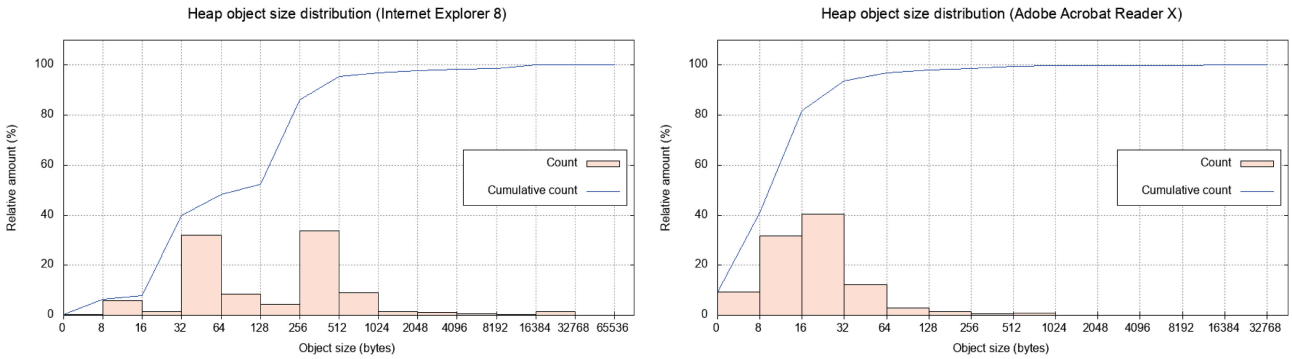


Fig. 7. Size distribution of allocated heap objects in Internet Explorer 8 and Adobe Acrobat Reader X.

TABLE 4
Unaligned Access Penalty (in Running Time) of Various Memcpy Implementations for 1-Mbyte Buffer

CPU	OS	Compiler and Library Environment	Penalty	Alignment Handling	IfAA/IfUA
Intel i7-6700	Windows 10	VS2015 vcruntime140.dll (32bit)	5%	yes	REP/REP
Intel i7-6700	Windows 10	VS2015 (64bit)	15%	yes	SSE/SSE
Intel i7-4980	Windows 8.1	VS2010 msvcr100d.dll (32bit)	−20%	yes	SSE/REP
Intel i7-4980	Windows 8.1	VS2010 msvcr100d.dll (64bit)	0%	yes	SSE/SSE
Intel i7-4980	Ubuntu 15.10	GCC 5.2.1 glibc-2.21 (32bit)	0%	yes	SSE/SSE
Intel i7-4980	Ubuntu 15.10	GCC 5.2.1 glibc-2.21 (64bit)	0%	yes	REP/REP
Intel i7-4980	Ubuntu 14.04	GCC 4.8.4 glibc2.19 (32bit)	10%	yes	REP/SSE
Intel i7-4980	Ubuntu 14.04	GCC 4.8.4 glibc2.19 (64 bit)	5%	yes	SSE/SSE
Intel i7-4980	OSX El Capitan	Apple LLVM 7.3 (32bit)	0%	yes	SSE/REP
Intel i7-4980	OSX El Capitan	Apple LLVM 7.3 (64 bit)	5%	yes	REP/REP
Intel i7-3770	Ubuntu 16.04	GCC 5.4.0 glibc-2.23 (64bit)	0%	yes	REP/REP
Intel i5-3570	Fedora20	GCC 4.8.2 glibc-2.18 (64bit)	0%	yes	SSE/SSE
Intel i5-3570	Debian7.5	GCC 4.7.2 glibc-2.13 (64bit)	0%	yes	SSE/SSE
Intel i5-3570	FreeBSD9.1 (32bit)	GCC 4.2.1 bsdlibc (32bit)	100%	no	REP/REP
Intel i5-760	Ubuntu 12.04 Server	GCC 4.6.3 (32bit)	0%	yes	SSE/SSE
Intel i5-760	Ubuntu 12.04 Server	GCC 4.6.3 (64bit)	0%	yes	SSE/SSE
Intel i5-760	Windows 7	VS2010 msvcr100d.dll (32bit)	50%	yes	SSE/REP
Intel i5-760	Windows 7	VS2010 msvcr100d.dll (64bit)	0%	yes	SSE/SSE

"IfAA" stands for: Instruction used for Aligned Address. "IfUA" stands for: Instruction used for Unaligned Address.

cache lines. In general, large heap chunks are requested to allocate buffers that are often accessed by bulk-memory access APIs such as `memcpy`. To verify the detailed impact of byte-granularity heap randomization against memory intensive APIs and large buffers, we analyzed various versions of memcpy implementations and conducted experiments. In particular, we ran 100,000 iterations of 1-Mbyte memory copy operation using memcpy and compare the execution time between two cases: (i) source and destination addresses of 1-Mbyte buffer are word-aligned, (ii) source and destination addresses of the buffer is not word-aligned (byte-aligned). This experiment is *NOT* designed to measure the performance of unaligned access at the instruction level. Rather, the purpose of this experiment is to measure the performance impact and compatibility of unaligned access against bulk memory access APIs. Table 4 summarizes the results.

In Table 4, the performance penalty is negligible in most cases. However, a severe performance penalty is observed from the case of i5-3570 FreeBSD 9.1; and ironically, the case of i7-4980 Windows 8.1 shows negative performance penalty. The cause of such peculiar results can be explained by the fact that `memcpy` chooses a different version of implementation at runtime, depending on various parameters such as the address alignment, CPU features, size of the buffer, and so forth. Aside from the case of FreeBSD 9.1, all version of memcpy implementation detected the unaligned address and optimized the performance by changing the alignment to be aligned before beginning the actual memory access.

4.4 SPEC2006 Benchmark

To measure the performance impact and memory usage of ra-malloc under memory intensive environment, we use SPEC2006. The SPEC2006 benchmark suite did not suffer compatibility problem after we applied ra-malloc. However, measuring the performance impact of ra-malloc should be carefully conducted because existing software conventionally assume word granularity heap alignment, therefore some code could show unexpected behavior. In addition, the program might use multiple allocators or custom allocators thus render the experiment inaccurate.

Before applying ra-malloc allocator, we analyzed the source code of each benchmark suite to verify if the application is suited for the experiment. In case of 400.perlbench, custom heap allocator (`Perl_safemalloc`) was used depending on the build configuration parameters. However, we confirmed that under our build configuration, the benchmark used default glibc allocator (`malloc`). Similarly, 403.gcc had multiple custom allocators (`xmalloc` and

TABLE 5
SPEC2006 Benchmark Results

Benchmark	jemalloc	ra-malloc	ottomalloc	dl-malloc	tcmalloc	# Allocs
perlbench	259	263	330	264	252	358,141,317
bzip2	421	434	420	424	417	168
gcc	231	238	318	242	233	28,458,531
mcf	195	202	196	200	197	3
gobmk	405	413	407	403	404	656,924
hmmer	348	386	344	344	349	2,474,260
sjeng	413	423	406	407	407	4
libquantum	353	356	352	354	353	179
h264ref	440	529	441	440	447	177,764
omnetpp	211	217	284	270	218	267,158,621
astar	310	316	306	325	307	4,799,953
xalancbmk	130	145	228	172	122	135,155,546
Total (geomean)	94.8%	99.7%	106.5%	100.0%	94.3%	

(a) Execution Time (sec). Dl-malloc is default (GLIBC).

Benchmark	Default	ra-malloc
perlbench	648	631
bzip2	832	835
gcc	865	868
mcf	1638	1646
gobmk	29	37
hmmer	26	32
sjeng	172	178
libquantum	95	105
h264ref	64	80
omnetpp	168	177
astar	321	367
xalancbmk	413	517
Total (geomean)	100%	110.38%

(b) Memory consumption (MB)

Various allocators (including ra-malloc) are applied to each program. While other allocators respect word-alignment for all chunks, ra-malloc randomizes the chunk location with byte-granularity.

`obstack`) but we checked that under our build-environment, glibc allocator was used (`obstack` internally used `xmalloc`, `xmalloc` internally used `glibc` allocator). Other benchmark suites had no particular issues. All benchmark suite programs were also dynamically analyzed to confirm how many heap chunks are affected.

To change the allocator of SPEC2006 without manual code rewriting, we made an LLVM pass *ra-malloc* and used `replaceAllUsesWith` LLVM API to replace glibc C/C++ allocators such as `malloc`, `free`, `_Znwj`, and `_ZdlPv`. To measure the performance impact, we applied ra-malloc as well as other open-source allocators to SPEC2006. The open-source allocators we used are `dlmalloc` (glibc), `tcmalloc`, `jemalloc`, and `ottomalloc`. The benchmark was conducted the under following environment: Intel(R) Xeon (R) E5-2630 CPU with 128GB RAM, Linux 4.4.0 x86-64 and Ubuntu 16.04. We used the glibc allocator as the baseline of performance. Table 5 summarizes the results. The average memory access overhead of ra-malloc is less than 5 percent considering it is based on je-malloc. Naive adoption of byte-granularity heap randomization against default libc heap imposes more than 10 percent memory access overhead.

5 Compatibility Analysis of Byte-Granularity Heap Randomization

Modern software and system implementations often assume and require the heap allocation alignment to be word aligned. Breaking this assumption can affect existing software in various aspects. In order to adopt byte-granularity heap randomization as practical defense, implementation conflicts regarding heap alignment should be addressed. To analyze the compatibility issues of byte-granularity heap randomization, we conducted experiments with various applications/benchmarks. Table 7 summarizes the analysis results regarding various compatibility issues.

5.1 Coreutils

After substituting Coreutils [47] glibc allocator to ra-malloc, we ran the Coreutils test suite to see if there are compatibility issues. In our initial experiment, 56 out of 476 Coreutils test cases did not pass the test. It turned out that programs using the following APIs crashed during execution: `strdup`, `strndup`, `getline`, `getdelim`, `asprintf`, `vasprintf`, `realpath`, `getcwd`. After analyzing the root cause, we found that the reason was irrelevant to the byte-level allocation granularity. While our LLVM pass replaces the allocator calls in Coreutils, allocator calls inside libc remained. Since the above-mentioned libc APIs internally use default glibc allocator, two allocators have conflicted. To handle this issue, we ported such APIs to use ra-malloc allocator and extended the LLVM pass to replace such API calls as well. After porting the above-mentioned APIs for ra-malloc, all programs passed the test without causing any compatibility issues.

5.2 Nginx

We ran Nginx test suite after applying ra-malloc allocator. In our initial experiment, none of the 324 `nginx-tests` suite passed the check. According to our analysis, the root cause of the problem was the implicit assumption of heap chunk alignment in Nginx implementation. Because the code assumes that any heap object will be word-aligned, on some occasions the program chose to store boolean information inside the least significant bit (LSB) of pointers, perhaps for performance reason. That is, instead of declaring a boolean member in a structure or class, the boolean value is saved in the LSB of the object pointer.

To handle this compatibility problem, we patched the Nginx-1.13.12 source code as shown in Listing 1. After the patch, all test suite passed the check. To evaluate the performance impact of ra-malloc-compatible modification against Nginx, we benchmarked the request throughput using `wrk` [50]. The benchmark was repeated 10 times for each case. Table 6 summarizes the benchmark result. According to the benchmark, no significant performance degradation was introduced by the modification.

TABLE 6
Performance Impact of the Nginx Patch

	requests/sec	
	Average	Standard deviation
Original	31362.20	565.90
Patched	31201.65	506.24

TABLE 7
Compatibility Analysis Summary

Application	# failures	Remarks
Coreutils	0/476	56 cases initially failed due to allocator substitution problem (fixed later)
Nginx	324/324	All cases failed due to least significant bit (LSB) of pointer utilization issue
Nginx (patched)	0/324	Re-factorization of LSB pointer issue made Nginx fully compatible
ChakraCore	172/2,638	Alignment Fault while using SSE (MOVAPS)
ChakraCore	176/2,638	Assertion failure (alignment check)
ChakraCore	3/2,638	Futex system call failure (unaligned parameter)

5.3 ChakraCore

ChakraCore revealed interesting compatibility issues of byte-granularity heap randomization. We applied ra-malloc to ChakraCore then ran the standard test suite provided by the ChakraCore. The initial result indicated that all test cases failed to pass the check with the same error. The failure seemed irrelevant to the test case. We found that the initialization of ChakraCore JSRuntime accessed ra-malloc-affected chunk, however, the `-O3` optimization of Clang aggressively assumed the alignment of the heap chunk (assuming it would be 128-bit aligned) then used SSE instructions that require specific memory alignment such as `movaps` for fast execution. After changing the optimization level to `-O0`, 351 over 2,638 test cases (`interpreted variant`) failed to pass the check.

Listing 1. Representative Parts of the Nginx Patch. LSB Storage is Replaced by a New Member in `ngx_connection_s`

```
diff -git a/src/core/ngx_connection.h b/src/core
    /ngx_connection.h
index e4dfe58..5c15ca2 100644
-a/src/core/ngx_connection.h
+++ b/src/core/ngx_connection.h
@@ -119,6 +119,7 @@ typedef enum {
  struct ngx_connection_s {
+    unsigned           instance:1;
     void               *data;
     ngx_event_t        *read;
     ngx_event_t        *write;
diff -git a/src/event/modules/ngx_epoll_module.c
     b/src/event/modules/ngx_epoll_module.c
index 76aee08..da948f2 100644
-a/src/event/modules/ngx_epoll_module.c
+++ b/src/event/modules/ngx_epoll_module.c
@@ -618,7 +620,8 @@ ngx_epoll_add_event(ngx_
    event_t *ev, ngx_int_t event, ngx_uint_t flags)
#endif
ee.events = events | (uint32_t) flags;
- ee.data.ptr = (void *) ((uintptr_t) c | ev->
 instance);
+ c->instance = ev->instance;
+ ee.data.ptr = c;
ngx_log_debug3(NGX_LOG_DEBUG_EVENT, ev->log, 0,
            "epoll add event: fd:%d op:%d ev:%08XD",
@@ -836,8 +841,7 @@ ngx_epoll_process_events
    (ngx_cycle_t *cycle, ngx_msec_t timer,
    ngx_uint_t flags)
     for (i = 0; i < events; i++) {
         c = event_list[i].data.ptr;
-        instance = (uintptr_t) c & 1;
-        c = (ngx_connection_t *) ((uintptr_t) c &
   (uintptr_t) 1);
+       instance = c->instance;
        rev = c->read;
```

Among the failures, 172 cases were caused due to SSE instruction alignment fault and 176 cases failed due to the assertion failure that explicitly requires word alignment for heap pointers before further operation. Interestingly, three cases failed due to time-out. After further analysis, we found that the failure was due to `futex` system call failure. Unlike `pthread mutex`, which is based on user-level `lock` prefixed instructions, the `futex` is based on Linux kernel system call which requires word-aligned address for its parameter. Because of this, the test suite failed to operate correctly as the system call raised an error. After finding this issue, we investigated the current (4.x) Linux kernel system calls to find a similar case. The `futex` was the only one that caused the problem with byte-granularity heap randomization. Overall compatibility analysis indicates that byte-granularity heap randomization requires high deployment cost.

6 Limitation

ra-malloc for ARM. The performance impact of unaligned access is a serious issue for RISC processors such as ARM. In general, unaligned memory access is strongly discouraged in RISC architectures.[7] In fact, ARM architecture started to support hardware-level unaligned access for some instructions (e.g., `LDR`) since ARMv6 [42]. To investigate the feasibility of byte-granularity heap randomization in ARM, we conducted per-instruction memory access benchmark against Cortex-A9 and Cortex-A17. Table 8 summarizes the result. According to our analysis, ARM architecture since ARMv6 indeed supports hardware level unaligned access. However, the support is for only two memory access instructions (`LDR`, `STR`) and high-performance penalty is observed at every 8-byte address border regardless of L1 cache or page size. Instructions such as `LDM` shows over 10,000 percent performance penalty for unaligned access. The reason for such high penalty is due to the lack of hardware support. Since the hardware is incapable of executing `LDM` with unaligned memory operand, hardware raises fault signal and kernel emulates the instruction. In case of the `VLDR`, even emulation is not supported by the kernel. Therefore the execution fails on unaligned memory operand. Most importantly, unaligned memory operand does not support `LDREX` instruction which is required for instruction level atomicity. For such reasons, ARM based system is inappropriate to consider byte-granularity heap randomization at this point.

Brute Forcing Attack. Byte-granularity heap randomization hinders the use of crafted pointer thus hinder exploits which depends on crafted pointers. The use of each crafted

7. PowerPC architecture can have 4,000 percent penalty in the worst case of unaligned access [28].

TABLE 8
Per-Instruction Benchmark Against ARM CPUs

Architecture	Instruction	Penalty	Remarks
Cortex-A9	LDR/STR	100%	penalty occurs at 8-byte border
Cortex-A9	LDRB/STRB	0%	no penalty
Cortex-A9	LDM/STM	over 40,000%	penalty always occurs, kernel emulation
Cortex-A9	LDM/STM (ThumbEE)	7,000%	penalty always occurs, kernel emulation
Cortex-A9 (,A17)	VLDR/VSTR	N/A	alignment fault
Cortex-A9 (,A17)	LDREX/STREX	N/A	alignment fault (no atomicity)
Cortex-A17	LDR	100%	penalty occurs on 8-byte border
Cortex-A17	STR	50%	penalty occurs on 8-byte border
Cortex-A17	LDM/STM	over 2,000%	penalty always occurs, kernel emulation

The benchmark methodology is same to the Intel version.

pointer involves 75 percent of defense probability in a 32-bit system and 87.5 percent in case of a 64-bit system. The defense probability will exponentially increase in case several crafted pointers are independently required for the overall attack. However, even such probability is insufficient to stop *repeatable* brute-forcing attack. In general, an invalid pointer dereference raises segmentation fault and crashes the program therefore attack is not repeatable. However, according to a previous work [48], there are special cases in which program survives even under segmentation fault and allows to repeat the attack [9].

Side Channel Attack. With byte-granularity heap randomization, heap pointers do not follow word-granularity. In average, 75 percent of heap pointers are not word-aligned. Assuming if the attacker is somehow able to measure the performance of dereferencing heap pointers precisely, she might be able to tell that some of them are misaligned around particular memory border. For example, the attacker can guess that a heap pointer is spanning across page boundaries while being unaligned if the access speed is relatively slow. So far, we fail to find any useful attack scenario by identifying such pointers. However, in theory, this can be considered as a potential side channel attack against byte-granularity heap randomization.

Implementation Conflicts. The adoption of byte-granularity heap randomization creates various implementation conflicts as discussed in Section 5. However, one of the correctness problem among them is the use of LSB portion of heap pointer assuming the pointer is word-aligned. In Section 5, we used Nginx for discussion; however we also found this issue in other applications as well. For example, Internet Explorer 11 uses the same implementation approach to mark the chunk type (Isolation Heap). Any programming techniques that rely on the assumption that *the heap chunk has specific alignment* cannot be applied with byte-granularity heap randomization at the same time. In addition, `futex` is currently incompatible with ra-malloc as it requires a word-aligned address (other 4.x Linux system calls are not affected by alignment). Admittedly, the implementation compatibility issues are the major limitation for adopting byte-granularity heap randomization in practice as it requires significant engineering effort. However, we believe this is not a fundamental limitation that undermines the worth our research.

Information Disclosure Using Byte-Shift-Independent Non-Pointer Values. Byte granularity heap randomization imposes difficulty of hijacking pointers by breaking the `sizeof (void*)` allocation granularity of randomized chunk allocation. As the result of byte granularity randomness, an attacker cannot leverage *pointer spraying* technique for bypassing the randomized memory layout. The only option for reliable attack (other than information disclosure) is to rely on byte-shift-independent values, which make it hard (if not impossible with careful heap management) to craft valid pointers. But this is not the case for *byte-shift-independent non-pointer values*, which can allow an attacker to craft reliable memory corruption as intended and then escalate the attack further. A representative example would be string length corruption [3], [10] mentioned earlier in the paper Section 2.

7 Related Work

HeapTaichi. [43] shows various heap spraying techniques that leverage the allocation granularity of memory allocators. For example, if allocation granularity is fixed, an attacker can split nop-sleds into several pieces and stitch them together with jump instructions. HeapTaichi claimed that reduced allocation granularity in heap randomization is better for security. However, the minimal allocation granularity considered in HeapTaichi is pointer-width. Although HeapTaichi discussed in-depth heap allocation granularity issues, no discussion regarding byte-granularity allocation and its ramification regarding security/performance was made.

Address Space Layout Permutation. [58] adopts a high degree of randomness compared to the existing ASLR and also performs fine-grained permutation against the stack, heap, and other memory mapped regions. Heap randomization is not the main theme of this work. However, the paper includes descriptions regarding fine-grained heap randomization. To adopt fine-grained address permutation for a heap, a random (but page-aligned) virtual address between `0` and `3` GB (assuming 32bit memory space) is selected for the start of the heap. Afterwards, a random value between `0` and `4` KB is added to this address to achieve sub-page randomization. According to this method, heap pointers should have random byte-level alignment, which involves unaligned access problem. However, discussion regarding unaligned access (due to byte-level randomization) or the security effectiveness of byte-granularity randomization was not discussed despite ASLP covered a broad range of fine-grained randomization issues.

Address Space Randomization. [49] introduced fine-grained Address Space Randomization (ASR) for various OS-level components including heap object allocation. In this work,

heap layout is effectively randomized by prepending random size padding for each object and permuting the sequence of allocated objects. The paper comprehensively explores various memory layout randomization strategies and propose various ideas regarding live re-randomization. They implement each randomization policies by patching the kernel and dynamic loader, or using binary code translation techniques. However, the security and performance impact regarding byte-level memory layout randomization is not the main interest of the paper. The main focus of the paper is comprehensive OS-level ASR and live re-randomization with a minimal performance penalty.

Data Structure Randomization. Data Structure Layout Randomization (DSLR) [59] randomizes the heap object layout by inserting dummy members and permuting the sequence of each member variables inside an object at compilation time. The size of randomly inserted garbage member variable is multiple of `sizeof(void*)` thus respecting CPU alignment. The goal of DSLR is to diversify the kernel object layouts to hinder the kernel object manipulation attack performed by rootkits; in addition to thwarting the system fingerprinting and kernel object manipulation attack which relies on object layout information.

METAlloc. In general, heap exploitation can be broadly divided into two categories: (i) exploitation based on corrupting application data inside heap, and (ii) exploitation based on corrupting metadata of heap allocator. While we focus on the first case (heap exploits caused by application data), there are other works which are more focused on the latter issue. (e.g., A metadata management scheme [53] is based on a novel memory shadowing technique.) While ra-malloc uses diverse alignment to handle the cache border access, METAlloc enforces the uniform alignment to optimize metadata lookup performance.

Cling. The isolation heap protection approach separates the heap into the independent area so that objects are allocated at different parts depending on their types. Indeed, these approaches can be observed in both academia and industry. Cling [37] identifies the call site of a heap chunk allocation request by looking into the call stack. If the chunk allocation request originates from the same call site, Cling considers the type of heap chunk to be the same, which indicates that it is safe to reuse the same heap area for those chunk requests. If the type is assumed to be different from two allocation requests, Cling does not allow the heap area to be reused between those requests. In practice, the heap isolation methods can frequently be observed in various security-critical software such as Internet Explorer, Chrome, and Adobe Flash [31].

Other Heap Randomization Approaches. Incorporating randomization into heap allocation has been discussed in numerous previous works. Some approaches, such as those of Bhatkar et al. and Qin et al., respectively randomize the base address of the heap, as shown in [39], [61]. Others randomize the size of the heap chunks, word-granularity location, allocation sequence and so forth [38], [40], [53], [56], [57], [60], [66]. From all these heap fortifications works including our paper, the purpose in adopting the notion as well as the implementation differs from each other. The advancement from previous works is that we show how byte-granularity heap randomization mitigates crafted pointer spray, then design an allocator that optimizes performance cost of byte-granularity heap randomization. Instead of locating the heap chunk at memory location unpredictable by an attacker, byte granularity heap randomization aim to obstruct heap exploits which require pointer-width allocation granularity.

8 Conclusion

In this paper, we show an in-depth discussion of byte-granularity heap randomization. At first, breaking the randomization granularity from word to byte can be considered trivial. However, this seemingly insignificant change in granularity opened up a surprising number of research issues regarding exploit mitigation, performance, and deployment. Our security analysis based on 20 real-world heap attacks demonstrated the effectiveness of byte-granularity heap randomization in various circumstances. After the security discussion, we show an in-depth performance analysis of byte-granularity randomization. Based on the performance analysis, we designed and implemented ra-malloc: an allocator optimized for byte-granularity heap randomization. The design of ra-malloc leverages recent advancement of post-Nehalem Intel architectures for handling the misaligned access. In particular, ra-malloc considers specific allocation sites regarding cache line border. The compatibility analysis in this paper is based on ChakraCore, Nginx, and CoreUtils benchmarks. Unfortunately, compatibility analysis of byte-granularity heap randomization shows negative results.

Acknowledgments

This research was supported by National Research Foundation of Korea (NRF-2017R1A2B3006360), Office of Naval Research (N00014-18-1-2661), and IITP (Institute for information & communications Technology Promotion, IITP-2017-0-01853-003)

References

[1] CVE-2012–4792, 2012. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-4792. IE Use-After-Free Vulnerability
[2] CVE-2013–0025, 2013. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-4792 IE Use After Free Vulnerability
[3] CVE-2013–0634, 2013. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0634. Flash ByteArray Out-of-bound
[4] CVE-2013–0912, 2013. [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=180763. V8 Type Casting Error
[5] CVE-2013–2729, 2013. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-2729. Integer overflow in Adobe Reader
[6] CVE-2014–3176, 2014. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-3176. V8 vulnerability
[7] CVE-2015–1234, 2015. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-1234. Chrome buffer overflow
[8] CVE-2015–2411, 2015. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-2411. Internet Explorer Memory Corruption Vulnerability
[9] CVE-2015–6161, 2015. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-6161. Microsoft Browser ASLR Bypass
[10] CVE-2015–8651, 2015. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-8651. Flash Vector Out-of-bound Access

[11] CVE-2016–0175, 2016. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-0175. Win32k Information Leak
[12] CVE-2016–0189, 2016. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-0189. IE heap corruption
[13] CVE-2016–0191, 2016. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-0191. Scripting Engine Memory Corruption Vulnerability
[14] CVE-2016–0196, 2016. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-0196. Win32k Vulnerability
[15] CVE-2016–1016, 2016. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-1016. Use-after-free in Flash
[16] CVE-2016–1017, 2016. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-1017. Use-after-free in Flash
[17] CVE-2016–1653, 2016. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-1653. Windows OLE Vulnerability
[18] CVE-2016–1665, 2016. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-1665. Information leak in V8
[19] CVE-2016–1677, 2016. [Online]. Available: . Type confusion in V8
[20] CVE-2016–1686, 2016. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-1686. Out-of-bounds read in PDFium
[21] CVE-2016–1796, 2016. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-1796. OSX heap corruption
[22] CVE-2016–1857, 2016. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-1857. Heap corruption in safari
[23] CVE-2016–1859, 2016. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-1859. Use-after-free in Safari
[24] CVE-2016–5129, 2016. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-5129. Memory corruption in V8
[25] CVE-2017–0071, 2017. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2017-0071. Chakra Heap Corruption
[26] CVE-2017–2521, 2017. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2017-2521. Webkit Out-of-bound Access
[27] CVE-2017–5030, 2017. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2017-5030. Out-of-bounds in V8
[28] Developer Works, I. Data alignment: Straighten up and fly right, 2015. [Online]. Available: https://www.ibm.com/developerworks/library/pa-dalign/
[29] ECMAScript language specification, 2018. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf
[30] Google Project Zero, 2018. [Online]. Available: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html
[31] Isolated heap for internet explorer, 2014. [Online]. Available: http://blog.trendmicro.com/trendlabs-security-intelligence/isolated-heap-for-internet-explorer-helps-mitigate-uaf-exploits/
[32] MITRE, 1999. [Online]. Available: https://cve.mitre.org. Common vulnerabilities and exposures
[33] MWR LABS PWN2OWN 2013 Write up, 2013. [Online]. Available: https://labs.mwrinfosecurity.com/blog/2013/04/19/mwr-labs-pwn2own-2013-write-up—webkit-exploit
[34] Performance Counters for Linux, 2008. [Online]. Available: http://lwn.net/Articles/310176/
[35] Streaming SIMD Extensions, 2009. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Streaming_SIMD_Extensions&oldid=704130503
[36] What is good memory corruption, 2015. [Online]. Available: https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html
[37] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers," in *Proc. USENIX Security Symp.*, 2010, pp. 177–192.
[38] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic memory safety for unsafe languages," *ACM SIGPLAN Notices*, vol. 41, pp. 158–168, 2006.
[39] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proc. 12th Conf. USENIX Security Symp.*, 2003, pp. 8–8.
[40] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Efficient techniques for comprehensive protection from memory error exploits," in *Proc. 14th Conf. USENIX Security Symp.*, 2005, pp. 17–17.
[41] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Proc. IEEE Symp. Security Privacy*, 2014, pp. 227–242.
[42] D. Brash, The ARM Architecture Version 6 (ARMv6), Jan. 2002, ARM White Paper, ARM Ltd., Cambridge, U.K.
[43] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou, "Heap Taichi: Exploiting memory allocation granularity in heap-spraying attacks," in *Proc. 26th Annu. Comput. Security Appl. Conf.*, 2010, pp. 327–336.
[44] C. Lewis and J. Tammariello, "Creating centralized reporting for microsoft host protection technologies: The enhanced mitigation experience toolkit," Carnegie-Mellon Univ Pittsburgh, PA, USA, 2016.
[45] I. Evans et al., "Missing the point (er): On the effectiveness of code pointer integrity," in *Proc. IEEE Symp. Security Privacy*, 2015, pp. 781–796.
[46] A. Fog, "The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers," *Copenhagen Univ. College Eng.*, pp. 2–29, 2012.
[47] Free Software Foundation, "Coreutils," 2016. [Online]. Available: http://www.gnu.org/software/coreutils/coreutils.html
[48] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz, "Enabling client-side crash-resistance to overcome diversification and information hiding." in *Proc. Annu. Netw. Distrib. Syst. Security Symp.*, 2016.
[49] 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.*, 2012, pp. 475–490.
[50] W. Glozer, "Modern HTTP benchmarking tool," 2014. [Online]. Available: https://github.com/wg/wrk
[51] A.-A. Hariri, S. Zuckerbraun, and B. Gorenc, "Understanding weaknesses within Internet Explorer's Isolated Heap and MemoryProtection," *Black Hat USA*, 2015.
[52] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2017, Art. no. 13.
[53] I. Haller, E. Van Der Kouwe, C. Giuffrida, and H. Bos, "METAlloc: Efficient and comprehensive metadata management for software security hardening," in *Proc. 9th Eur. Workshop Syst. Security*, 2016, Art. no. 5.
[54] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *Proc. IEEE Symp. Security Privacy*, 2013, pp. 191–205.
[55] Intel. Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3A: System Programming Guide, Part, vol. 1, no. 64, p. 64.
[56] V. Iyer, A. Kanitkar, P. Dasgupta, and R. Srinivasan, "Preventing overflow attacks by memory randomization," in *Proc. IEEE 21st Int. Symp. Softw. Rel. Eng.*, 2010, pp. 339–347.
[57] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic, "Comprehensively and efficiently protecting the heap," *ACM SIGPLAN Notices*, vol. 41, pp. 207–218, 2006.
[58] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Proc. 22nd Annu. Comput. Security Appl. Conf.*, 2006, pp. 339–348.
[59] Z. Lin, R. Riley, and D. Xu, "Polymorphing software by randomizing data structure layout," in *Proc. Int. Conf. Detection Intrusions Malware Vulnerability Assessment*, 2009, pp. 107–126.
[60] G. Novark and E. D. Berger, "DieHarder: Securing the heap," in *Proc. 17th ACM Conf. Comput. Commun. Security*, 2010, pp. 573–584.
[61] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating bugs as allergies—A safe method to survive software failures," *ACM SIGOPS Operating Syst. Rev.*, vol. 39, pp. 235–248, 2005.
[62] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos, "VTPin: Practical VTable hijacking protection for binaries," in *Proc. 32nd Annu. Conf. Comput. Security Appl.*, 2016, pp. 448–459.
[63] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2014, pp. 54–65.

[64] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proc. IEEE Symp. Security Privacy*, 2013, pp. 574–588.
[65] M. E. Thomadakis, "The architecture of the Nehalem processor and Nehalem-EP SMP platforms," 2011. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.455.4198&rep=rep1&type=pdf
[66] C. Valasek and T. Mandt, "Windows 8 heap internals," *Black Hat USA*, 2012.
[67] R. Wojtczuk, "TSX improves timing attacks against KASLR," 2014.
[68] W. Zhong, "Dice: A nondeterministic memory alignment defense against heap Taichi," PhD thesis, Dept. Comput. Sci. Eng., Pennsylvania State Univ., State College, PA, USA, 2011.

Daehee Jang received the BS degree in computer engineering from Hanyang University, South Korea, in 2012, and the MS and PhD degrees in information security from the Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2014 and 2019, respectively. He is currently a post doctoral researcher with Georgia Tech. His current research interests include operating system, memory safety, and fuzzing.

Jonghwan Kim received the BS degree in computer engineering from the University of Electro-Communications, Japan, in 2011, and the MS degree in information security from the Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2014. He is currently working toward the PhD degree in the Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST). His research interests include software vulnerability, operating system.

Hojoon Lee received the BS degree from the University of Texas at Austin, and the MS and PhD degrees from KAIST. He is currently an assistant professor with the Department of Computer Science Engineering, Sungkyunkwan University, Korea. His current research interests include but not limited to trusted execution environments, software security, and cloud security.

Minjoon Park received the BS and MS degrees from the Graduate School of Information Security and School of Computing, Korea Advanced Institue of Science and Technology (KAIST). He is currently working toward the PhD degree in the Graduate School of Information Security, KAIST, South Korea. His research interests include system and software security, in particular protection systems through formal verification.

Yunjong Jung received the BS degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST). He is currently working toward the MS degree in the Graduate School of Information Security, Korea Advanced Institute of Science and Technology, South Korea. His research interests include systems and software security, in particular protection of software using trusted execution environments (TEEs).

Minsu Kim received the BS degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), and the MS and PhD degrees in information security from the KAIST, South Korea. He is currently a senior researcher with the National Security Research Institute, South Korea. His research interests include system security, especially in the mitigations against code reuse attack (CRA).

Brent Byunghoon Kang received the BS degree from Seoul National University, the MS degree from the University of Maryland at College Park, and the PhD degree in computer science from the University of California at Berkeley. He is currently the chief professor with the Graduate School of Information Security, and associate professor with the School of Computing, KAIST. He has also been with George Mason University as an associate professor in the Volgenau School of Engineering. He has been working on systems security including OS kernel integrity monitors, HWbased trusted execution environment, code-reuse attack defenses, memory address translation integrity, and heap memory defenses. He is a member of the IEEE.