ICIC Express Letters
Part B: Applications
Volume **8**, Number **3**, March **2017**

 ISSN 2185-2766
pp. 587–592

EFFICIENT DANGLING POINTER DETECTOR FOR C/C++ PROGRAMS

Zhenyu Yang

Information Center
China Tobacco Guangxi Industrial CO., LTD.
No. 28, Beihu South Road, Xixiangtang District, Nanning 530001, P. R. China
zhenyuyang2016@163.com

Received September 2016; accepted December 2016

Abstract. *Dangling pointer error is a notorious memory error in C/C++ programs. It is hard to debug and is often leveraged to compromise the security of systems. This paper introduces an efficient runtime system to detect all dangling pointer error in C/C++ programs. Our work is based on a key finding: objects allocated in the same execution context are likely to be freed together. Based on this insight, we propose a new heap design and a new kernel page fault handler to detect accesses to already-freed objects. Compared with previous state-of-the-art work (introduces up to 21X overhead), our detector introduces acceptable runtime overhead (with 34%) and consumes less physical memory on allocation intensive benchmarks.*
Keywords: Dangling pointer error, Execution context, Heap design, Kernel page fault handler

1. **Introduction.** Dangling pointer error (accessing already freed memory location) is a representative memory error in C/C++ programs. On the one hand, it is notoriously hard to debug, limiting the productivity of programmers [1]. On the other hand, it degrades the reliability and security of systems and in many cases it is exploited to achieve some kind of attacking (double free vulnerability in SQL [2]).

Detecting dangling pointer error is an old topic. However, previous work all falls short on some aspects. Static analysis methods [3] are often trapped by high ratio of false-positives. Runtime detecting methods [1] can detect all dangling pointer errors but normally introduce large overhead. For example, the famous Valgrind [4] tool which is used to check all memory errors usually introduces over 8-10X overhead due to its mechanism (dynamic instrumentation [5]) to trap every load and store instruction. Instrumenting all memory accesses at compiling time also introduces over 3-4X overhead [6,9,10]. The state-of-the-art work [1] gives each object a single virtual page and leverage page fault mechanism to detect dangling pointer error. It exhibits good performance on some small memory-footprint applications. However, when applied to any application that allocates large amount of objects or memory, it introduces unacceptable overhead (for the Olden benchmark suite it introduces up to 11X overhead [1]) due to virtual memory explosion and tlb miss (see details in Section 2). Overall, previous tools are not practical to be adopted in production environment because of their high runtime overhead. A high efficient mechanism is thus desirable.

This paper introduces a high-efficient detector for dangling pointer error. Our method is based on a key insight: the objects allocated in the same context are likely to be freed in the same context together. We will demonstrate in this paper (see Section 2) that most objects allocated in the same call stack (context) are freed together, indicating a better heap design to put all these objects into the same virtual page. Thus, if one object is freed, we just set the corresponding virtual page (the object resides in) to be unaccessible (in

page table). Further accesses to the already-freed object will trigger page fault and we can determine whether this is a dangling pointer error. For other accesses to the valid objects in the same virtual page, we argue that this situation seldom happens because according to our findings, they are likely to be freed together and are not likely to be accessed after one of them is freed. However, this situation happens with certain possibility. For this we introduce a memory access monitoring method based on page fault mechanism: if upper applications access one valid object in a virtual page which has already being set un-accessible, a page fault will be triggered. In the page fault handler we will let the kernel perform the memory access for the application and we do not change the state of the virtual page (see detail in Section 3). We maintain the virtual page to be un-accessible and thus we are able to trap further memory accesses to this page and detect all dangling pointer errors. Experiments show that our work can detect all dangling pointer errors with little overhead (within 34%) and is much better than the state-of-the-art work (up to 11X overhead) on allocation intensive benchmarks [1].

The organization of the rest of this paper is as follows. We introduce background and our key findings in Section 2. We describe the design of our method in Section 3. Section 4 shows experimental results and Section 5 concludes this paper.

2. **Background and Motivation.** Memory access monitoring is the core to detect dangling pointer error. Previous studies all focus on trapping and examining every memory access in order to check if it accesses already-freed objects. Dynamic and compiler instrumentation [5,6] inserts examination code before every memory access instruction to perform memory access monitoring and is too heavy (3-10X overhead). Current state-of-the-art work [1] is more efficient as it is based on the paging mechanism supported by the underlying hardware.

The idea of the state-of-the-art work is shown in Figure 1. When serving the upper applications' object allocation calls (malloc in C), it gives each object a single virtual page and maps these virtual pages into the same physical page. The one-object-one-virtual-page mechanism greatly facilitates monitoring memory accesses to the objects. For example, if the upper application frees the object B (calls free(B)), it then sets the corresponding virtual page to be un-accessible in the page table. Then if any instruction accesses the object B, it reports that a dangling pointer error has been detected.

This mechanism works fine for the applications that allocate small amount of object. However, for allocation intensive applications [1], it introduces unacceptable overhead due to virtual space explosion and tlb miss. For example, for the benchmark *espresso* [7], it allocates 10,000+ objects of small sizes (less than 64 bytes), consuming 1,000X more virtual spaces, leading to 1,000X more tlb miss and over 20X runtime slowdown. The main source of overhead is its mechanism to give each object a single virtual page.

Above all, a naïve way to optimize the state-of-the-art work is to allocate more objects on one virtual page to reduce virtual page usage. However, doing this will introduce other problems and overhead. For example, if objects A and B are in the same virtual page and

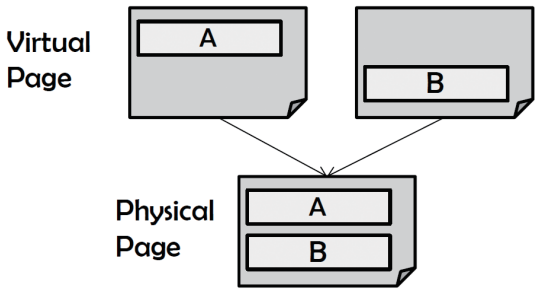

FIGURE 1. Previous many-to-one mapping method

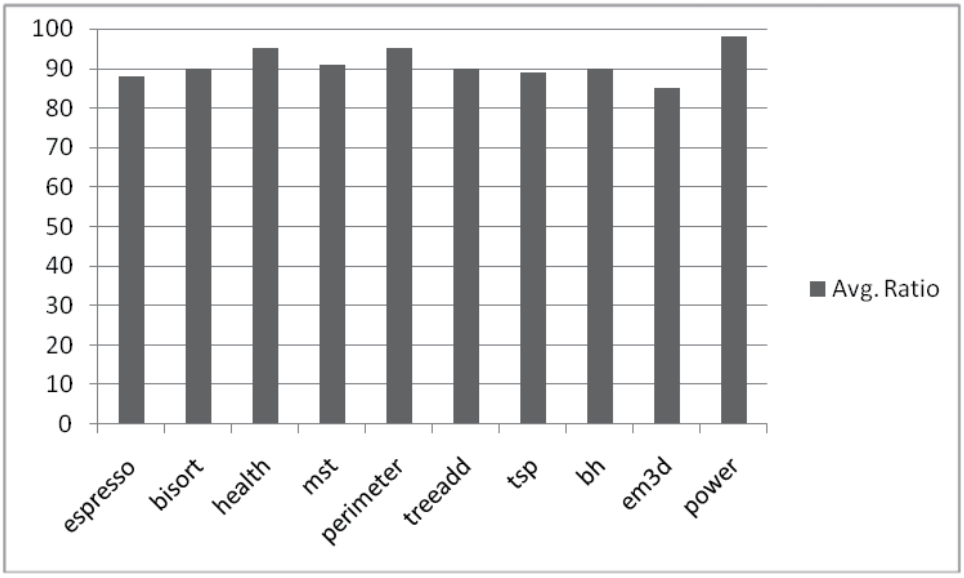

FIGURE 2. Average ratio

we just freed B, then the virtual page should be set un-accessible and thus we can catch further accesses to B (through page fault mechanism). However, we want to let normal accesses to A executed as usual. In order to achieve this, we introduce a mechanism that we let the kernel perform the memory access for the application (see detail in Section 3). This is done every time when object A is accessed and the page fault is triggered. Thus, here we introduce another main source of overhead: the page fault.

To reduce this kind of overhead (the page fault triggered when accessing valid objects), here we introduce our key findings that some groups of objects (allocated in the same context) may be safely put into the same virtual page and there is a high possibility that if one object in the virtual page is freed, other objects of the same group are likely to be freed soon and will not be accessed. Thus, the page fault is seldom triggered in our mechanism. Figure 2 shows our experimental results on benchmarks from Olden benchmark suite and the benchmark *espresso*. For example, for the benchmark *espresso*, the y axis means that 88% of objects which are allocated in the same context (call stack) are freed together in another same context. Thus, if we put these objects into the same virtual page, we can greatly reduce the page fault overhead and at the same time detect all the dangling pointer errors.

3. **Efficient Dangling Pointer Error Detector.**

This section gives our design of efficient dangling pointer error detector. Our design is based on the findings introduced in Section 2 and contains two parts: (1) a memory allocator (supports malloc and free) that dynamically collects context information and allocates objects according to their contexts; (2) a kernel page fault handler that lets memory accesses to valid objects go through un-accessible virtual pages.

3.1. **Memory allocator.**

We build our memory allocator based on the widely-adopted memory allocator Hoard [7,11,12]. When serving malloc from upper applications, a memory allocator would firstly ask a large piece of virtual memory (called chunk in this paper) from operating system (using mmap) and then return a proper piece to upper applications to use. Traditionally, memory allocators (such as Hoard) organize chunks into different classes. As shown in Figure 3(a), the 8-byte chunk only allocates objects of 8 bytes while the 64-byte chunk only allocates objects of 64 bytes. 1-16 bytes allocation will be severed in the 16-byte chunk and 17-32 bytes allocation will be severed in the 32-byte chunk and so on. This strategy is simple and can serve malloc and free at very high performance.

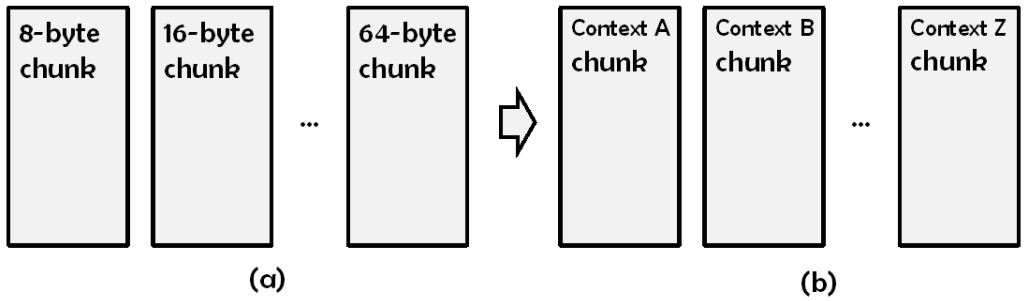

FIGURE 3. Heap organization

In our mechanism, as we need to put objects of the same context into the same virtual page, we organize the chunks according to different contexts as shown in Figure 3(b). For example, if currently we are in context A, then all the memory allocations are served in the Context A chunk. The objects may have different sizes. Here we use a bitmap to store the allocation information for different chunks. One bit in the bitmap represents 8 bytes. Lastly, we get the current context (call stack) using backtrace mechanism (the Linux backtrace function, man backtrace to see details). Above all, this chunk organization enables us to put objects of the same context in the same virtual page.

3.2. **Page fault handler.** In our mechanism we put several objects into the same virtual page. If one object is freed, we need to set the whole virtual page to be un-accessible (in page table) to catch further accesses to the freed object. However, further accesses to valid objects in the same virtual page must execute normally. A naïve idea to achieve this is when a page fault is triggered by accessing valid objects, we can first set the virtual page to be accessible and let the memory access go. Then after the access we set the virtual page to be un-accessible again to trap further accesses. The problem is, after we set the virtual page to be accessible, we are not able to trap the further accesses to the same page because further memory accesses to that virtual page will all execute without triggering any page fault.

To solve this problem, here we introduce a mechanism to let normal accesses run without compromising the ability to monitor accesses to freed objects. First, as shown in Figure 4, in our memory allocator we map two virtual pages to each physical page (this could be done at chunk allocation time. We use mmap to allocate a whole chunk from Linux kernel and we can use mmap to map two virtual chunks to the same physical chunk, just like inter-process memory sharing). For the two virtual pages, one is used in upper applications and may be set un-accessible to detect dangling pointer error. Another one is always accessible and will be used in our kernel. Thus, at anytime, the kernel could always use the second virtual page to access objects. Second, we perform the following operations when the upper application triggers page fault by accessing valid objects: (1)

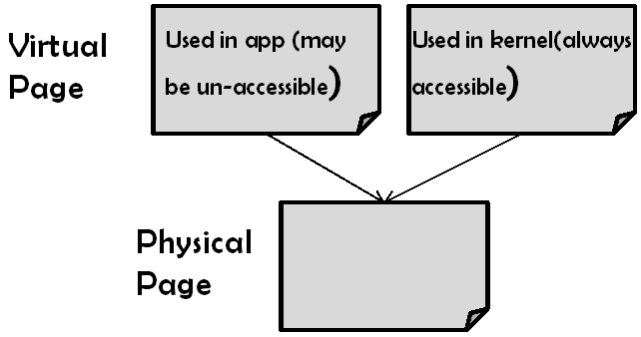

FIGURE 4. Access control by page mapping

get the instruction which triggers page fault through the ip counter. This instruction is a memory accessing instruction; (2) decode this instruction and do the memory accessing through the accessible virtual page (as shown in Figure 4). For example, if the instruction is *load eax, [addr]*, we will access the object of addr using the corresponding accessible virtual pages (the second virtual page as shown in Figure 4) and fetch the data into kernel stack (eax register on the kernel stack). The decoding and translation can be done by leveraging previous table-based binary translation tools (such as libdetox) [8]; (3) increase the ip counter to jump to the next instruction. Above all, the kernel could do the memory access for applications to access un-accessible pages without changing the state of the virtual pages, which allows us to catch further accesses to freed objects.

4. **Experimental Results.** This paper introduces a high-efficient detector for dangling pointer error (we call DDPE in this section). In experiment we mainly test and show its runtime overhead and compare it with the state-of-the-art work [1] (we call Pre in this section) to show its advantages on allocation intensive applications.

Our experimental platform is an AMD server (12-core opteron 7174) equipped with 64GB physical memory running Linux 3.11. The benchmark we use is Olden benchmark suite and *espresso* [7]. All the benchmarks allocate large amount of objects and virtual memory.

The experimental results are shown in Figures 5 and 6. Figure 5 shows the normalized execution time. We can see for all the benchmarks except *bh*, *em3d*, and *power*, Pre introduces obvious overhead, ranging from 2X to 21X. This is because of its design to give each object a single virtual page that causes virtual space explosion and more tlb miss. For the benchmark *bh*, *em3d*, and *power*, they allocate less objects and thus Pre achieves better performance. The DDPE represents our work. We can see for all the benchmarks, our DDPE achieves much better performance (with 34% overhead) than Pre because of our new design of heap. We can see the overhead of our mechanism (DDPE) is mainly determined by the object allocation and deallocation pattern of upper applications as discussed in Section 2 (see Figure 2).

Figure 6 shows the normalized memory overhead. Pre uses more physical memory because it needs more slot in page table to store the mapping information for increased virtual pages it uses for each object. Our DDPE uses more padding memory as we have to allocate object according to different allocation context. In all, our method is much

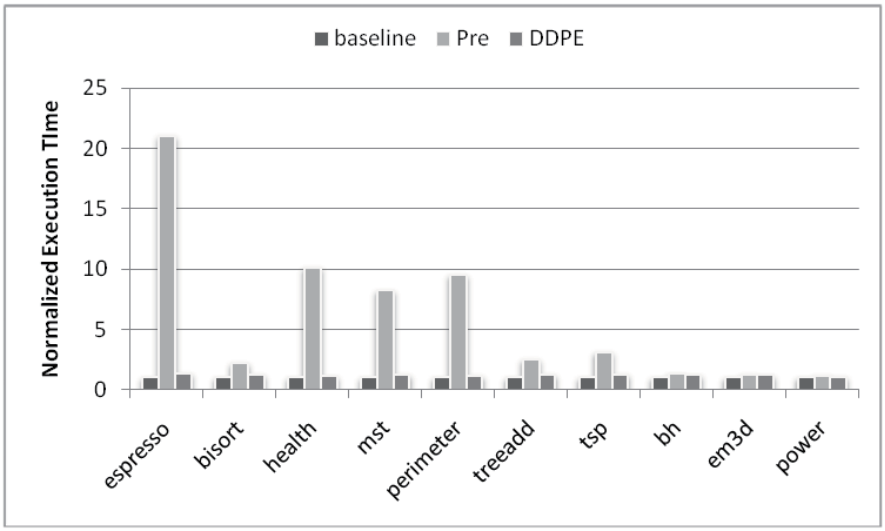

FIGURE 5. Experimental results of execution time (baseline is native execution, Pre is previous state-of-the-art work, and DDPE is our work)

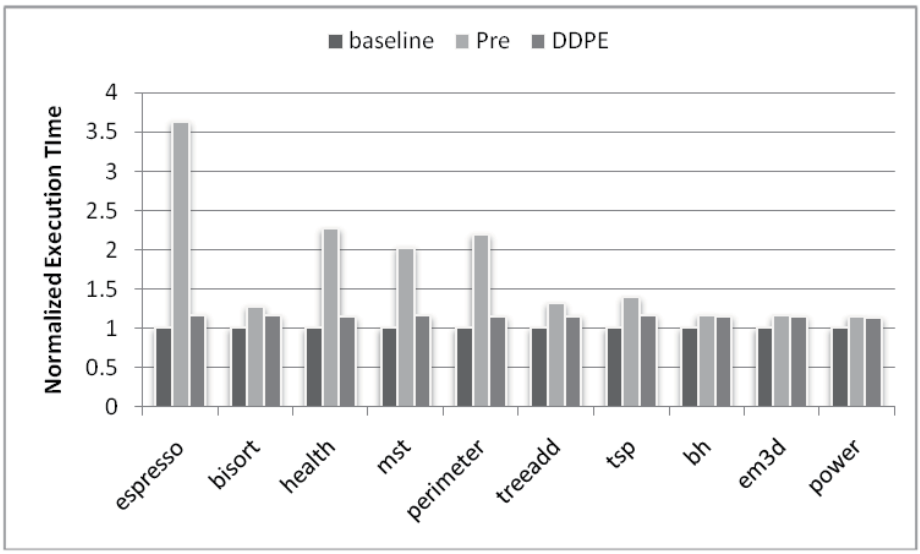

FIGURE 6. Experimental results of memory usage (baseline is native execution, Pre is previous state-of-the-art work, and DDPE is our work)

more efficient than previous state-of-the-art work on allocation intensive benchmarks and we consume less physical memory.

5. **Conclusions.** This paper introduces a high-efficient dangling pointer detector. Based on page fault mechanism and our key findings that objects allocated in the same context are likely to be freed together, we introduced a new heap design and a page fault handler that we can track all dangling pointer error very efficiently. Compared with previous state-of-the-art work (introduces up to 21X overhead on allocation intensive benchmarks), our detector introduces very small runtime overhead (within 34%) and shows great potential to be adopted in real production environment. Future work includes optimizing the memory allocator to better sever multi-threading.

REFERENCES

[1] D. Dinakar and V. Adve, Efficiently detecting all dangling pointer uses in production servers, *International Conference on Dependable Systems and Networks (DSN'06)*, 2006.
[2] Y. Younan, FreeSentry: Protecting against use-after-free vulnerabilities due to dangling pointers, *Network and Distributed System Security Symposium*, 2015.
[3] B. Lee et al., Preventing use-after-free with dangling pointers nullification, *Network and Distributed System Security Symposium*, 2015.
[4] N. Nethercote and J. Seward, Valgrind: A framework for heavyweight dynamic binary instrumentation, *ACM SIGPLAN Notices*, vol.42, no.6, 2007.
[5] D. Bruening, Q. Zhao and S. Amarasinghe, Transparent dynamic instrumentation, *ACM SIGPLAN Notices*, vol.47, no.7, pp.133-144, 2012.
[6] T. Iskhodzhanov, R. Kleckner and E. Stepanov, Combining compile-time and run-time instrumentation for testing tools, *Programmnye Produkty I Sistemy*, vol.3, pp.224-231, 2013.
[7] E. Berger et al., Hoard: A scalable memory allocator for multithreaded applications, *ACM SIGPLAN Notices*, vol.35, no.11, pp.117-128, 2000.
[8] B. Hawkins et al., Optimizing binary translation of dynamically generated code, *Proc. of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2015.
[9] J. Zhao et al., Formal verification of SSA-based optimizations for LLVM, *ACM SIGPLAN Notices*, vol.48, no.6, pp.175-186, 2013.
[10] C. Lattner and V. Adve, The LLVM compiler framework and infrastructure tutorial, *International Workshop on Languages and Compilers for Parallel Computing*, Springer Berlin Heidelberg, 2004.
[11] D. E. Berger, B. G. Zorn and K. S. Mckinley, Composing high-performance memory allocators, *ACM SIGPLAN Notices*, vol.36, no.5, pp.114-124, 2001.
[12] D. E. Berger, B. G. Zorn and K. S. Mckinley, OOPSLA 2002: Reconsidering custom memory allocation, *ACM SIGPLAN Notices*, vol.48, no.4S, pp.46-57, 2013.