# Efficiently Detecting Use-after-Free Exploits in Multi-Threaded Applications

Vinod Vijay Nigade
Vrije Universiteit Amsterdam
vinod.nigade@gmail.com

## Abstract

Many system applications are still developed using unsafe languages like C/C++. Memory corruption due to dangling pointers in these applications can lead to Use-after-Free (Double-Free) vulnerabilities. Many Use-after-Free detection schemes and systems have been proposed in the previous work. These systems either incur high performance overhead or have limited applicability. This paper proposes DangSan, a simple and efficient lock-less system to prevent Use-after-Free exploits during runtime. DangSan uses LLVM compiler instrumentation pass to insert run-time pointer tracking functions. During run-time, most of the pointers that point to the object are tracked. These pointer values are set to invalid addresses when the object is freed. DangSan is based on efficient metadata management framework, METAlloc [14]. It has moderate performance overhead. For SPEC2006 benchmarks, it has $43.9\%$ average (geometric mean) run-time overhead when only heap pointers are tracked. For widely used web-servers (`nginx`, `httpd` and `lighttpd`), it has moderate throughput degradation of $12.8\%$ and negligible service latency overhead. Moreover, DangSan successfully prevented recently discovered Use-after-Free vulnerabilities in widely used complex software.

## 1. Introduction

Many system applications are written in unsafe languages like C/C++. These languages are mostly used to have explicit control over hardware interfaces for optimal performance. For example, pointers are used to have explicit control on memory management. However, incorrect use of explicit control can lead to security vulnerabilities. Pointers incorrect use may lead to memory corruption. Use of dangling pointer is one instance of incorrect behaviour.

Dangling pointer is a pointer that points to the freed object. Use of dangling pointers i.e. Use-after-Free or Double-Free affects application integrity, availability and confidentiality [1]. Dangling pointer may write invalid data into newly allocated memory resulting into data corruption, thereby affecting integrity. Memory allocators consolidate two adjacent freed chunks into single big chunk. When Use-after-Free occurs after chunk consolidation, invalid data can be used as chunk information. This state results into free-list corruption that can lead to application crash, thereby affecting availability. Moreover, Use-after-Free before chunk consolidation is prone to arbitrary code execution, thereby affecting confidentiality. Segmentation fault due to Use-after-Free can leak memory addresses, thereby making Address Space Randomization (ASLR) protection weak [27]. Also, Use-after-Free vulnerabilities are reported and exploited in widely used browsers [2, 28]. Most of the highly critical vulnerabilities can be exploited with low complexity. The impact includes unauthorized information disclosure/ modification or service disruption [3].

Common defensive coding practice is to set dangling pointer to benign value `NULL`. Manually, this practice is not scalable in large code base when multiple pointer copies are present. Same technique can be used dynamically to track all pointers to the object and set pointer value to `NULL` when object is freed. State-of-the-art mitigation techniques like DangNull [19], FreeSentry [30] tracks pointer-object relationship dynamically (during run-time). Compiler infrastructure like CIL [21], LLVM [18] is used to insert run-time pointer tracking functions. DangNull uses red-black tree data structure to store and retrieve metadata (i.e. pointer-object relationship). It provides thread-safety for data structure operations using mutexes. However, DangNull incurs high average performance overhead of $80\%$. Also, it does not track stack and global pointers. FreeSentry has an average performance overhead of $25\%$. This performance is reported using CIL for static instrumentation. Performance numbers with LLVM are higher than CIL [6]. FreeSentry uses

[Efficiently Detecting Use-after-Free Exploits in Multi-Threaded Applications]

hash-table to store and retrieve metadata. However, thread-safety for data structure protection is missing in FreeSentry. Thus, it can break multi-threaded (production) applications. Also, FreeSentry has not evaluated throughput degradation of web-servers. Therefore, state-of-the-art mitigation techniques either incur high performance overhead or have limited applicability.

In dynamic analysis, pointer-object relationship is tracked. To store and retrieve relationship information, we need highly efficient and complete shadow memory management framework. It should have low lookup and memory overhead. METAlloc [14] is an efficient metadata management scheme. We first evaluated effectiveness of metadata management framework, METAlloc. We implemented FreeSentry scheme using METAlloc. We found that thread-safety (i.e. data structure protection) introduces huge performance overhead of $70\%$. Applications in production environment are highly multi-threaded. We have proposed simple and fast lock-less Use-after-Free detection scheme, called as DangSan to protect multi-threaded applications efficiently. DangSan maintains per-Thread and per-Object metadata, thereby it reduces thread synchronization. We implemented DangSan using METAlloc. DangSan has moderate run-time overhead of $43.9\%$ when only heap pointers are tracked. Moreover, it introduces only $4\%$ more overhead when all pointers are tracked (Stack, Heap and Global).

## 2. Background

Dangling pointer is created when the memory object is freed. Dangling pointer is exploitable only when it is accessed. Moreover, attacker needs control over the freed memory where dangling pointer points. Attacker can place desired data in the controlled freed memory. Based on the context in which dangling pointer is used, a particular exploit can be triggered [1, 3]. The time between dangling pointer creation and use is highly important for the exploit. Longer the time period, more the chances to exploit [5, 9]. Double free is a variation of use-after-free. Double free may corrupt memory allocator chunk information making system vulnerable. Dangling pointers are highly severe than spatial memory errors like, buffer overflow. Much of the research has happened to develop sophisticated buffer overflow mitigation techniques. Thus, spatial memory error vulnerabilities are hard to exploit. Due to this, dangling pointer vulnerabilities have gained popularity among attackers. However, mitigation techniques for dangling pointers are incomplete or incur high performance overhead.

Mitigation techniques include static analysis or run-time analysis. Static analysis on the source code or binary is hard [12]. It needs precise points-to analysis, type information, inter-procedural analysis. Moreover, object allocation, pointer propagation and object deallocation can occur at different places (functions, modules, threads) in the code. This further adds complexity to find accurate dangling pointers. On the other hand, dynamic analysis build pointer-object relationship during run-time. Pointer-object relationship is stored in object metadata (shadow object). Although, dynamic analysis is more accurate (low false positive and false negative rate) than static analysis, it incurs high performance overhead. Mostly, metadata lookups (i.e. finding metadata given an object or a pointer) are costly.

Dynamic analysis technique requires extra memory. This memory stores metadata associated with objects. Most of the schemes require object or pointer to metadata lookup. Metadata lookup should efficiently support object range lookups (i.e. finding metadata given any inbound object address). DangNull uses variant of red-black tree to store metadata. Tree node is a shadow object associated with an object. Every node stores root address and size information. This bound information is needed for fast range lookups. However, metadata lookup time is highly variable. That is, it depends on the height of the shadow tree. On the other hand, FreeSentry uses label based system [31]. Label based system store unique labels in the shadow memory for each fixed size object field. Object may have multiple fixed size entries in the label table. Each entry will have same label. During metadata lookup, unique label is searched in the label table. This unique label is used as index in the object lookup hash table. However, FreeSentry need to store more than one objects per entry in the object lookup table. Therefore, it may also have variable lookup time for the object metadata.

Furthermore, pointer to object metadata lookup is required during pointer propagation (i.e. tracking pointer information in object metadata). DangNull stores incoming and outgoing links in the shadow node along with bound information. Incoming link denotes that the object is pointed by other object and outgoing link denotes that the object is pointing to other object. DangNull retrieves shadow object representing pointer address. It checks and modifies outgoing link of this object to point to other shadow object (i.e. to the object pointed by the pointer). Similarly, it modifies incoming link of the other shadow object. It has variable metadata retrieval time. FreeSentry uses pointer lookup hash table to retrieve pointer information. Pointer lookup table does not require range lookups. Thus, hashtable is a valid choice. FreeSentry requires huge memory for label table (almost equal to process memory), object and pointer lookup tables. Moreover, shadow memory data structure has to be thread-safe. Large number of pointer propagations, object allocations and deallocations in multi-threaded application will drastically slow down application performance. Thus, synchronizing operations on data structures increase performance overhead. DangNull protects data structure using mutexes. However, FreeSentry has not focussed on thread-

safety. In FreeSentry, object and pointer lookup hash tables need to be protected. This can be achieved having a coarse-grain or a fine-grain per-hash table entry lock.

Any dangling pointer detection scheme (dynamic analysis) requires efficient metadata management (i.e. fast metadata allocation and retrieval strategy), thread-safety and low memory overhead. METAlloc [14] is an efficient and practical memory shadowing framework. It is based on the strategy that every object in the same memory page has same alignment. Thus, it maintains metadata information per page instead of per memory object. Given a pointer, it first finds corresponding page information i.e. metadata base address and alignment information. Next, it calculates pointer offset within the page. The offset and alignment information is used to find corresponding entry in the metadata area. Therefore, it has fixed metadata lookup time for any object address range. Moreover, it provides uniform metadata tracking for all the objects (Heap, Stack and Global). Most importantly, it incurs only $1.2\%$ average run-time performance overhead for SPEC2006 benchmarks. Low performance overhead and memory optimized design along with easy to use framework makes METAlloc an ideal choice for implementing dangling pointer prevention scheme.

We implemented FreeSentry scheme using METAlloc. METAlloc provides efficient object-to-metadata mapping. Therefore, we do not need Label table to fetch metadata associated with the objects. Object metadata stores a pointer to the pointer list (i.e. list of pointers information that are pointing to the object). Thus, we do not need object lookup table. Fetching a pointer information corresponding to an object requires fixed retrieval time. Next, we use pointer lookup hash table similar to FreeSentry. Pointer-to-object metadata is retrieved using pointer lookup table. Each entry in the pointer lookup table is a doubly list of pointer information. This pointer information is also a node in the object metadata list. Thus, given a pointer, fetching an object pointer list is just finding the correct pointer information in the pointer lookup table. We evaluated this design for SPEC2006 benchmarks. We performed experiments on $64$-bit CentOS Linux with Intel Xeon CPU E5-$2640$ v3. We implemented LLVM compiler pass to insert run-time pointer tracking function.

Figure 1 shows SPEC2006 run-time overhead for FreeSentry design using METAlloc. It depicts normalized (with baseline) numbers with and without thread-safety. With thread-safety, run-time overhead on an average (geometric mean) is $69.6\%$. We used pthread mutexes to protect object pointer list and pointer lookup table. Without thread-safety, run-time overhead is just $33.2\%$. This number matches with the average FreeSentry performance overhead when implemented on LLVM [6]. However, all our bechmarks are SPEC2006 whereas FreeSentry has mix of

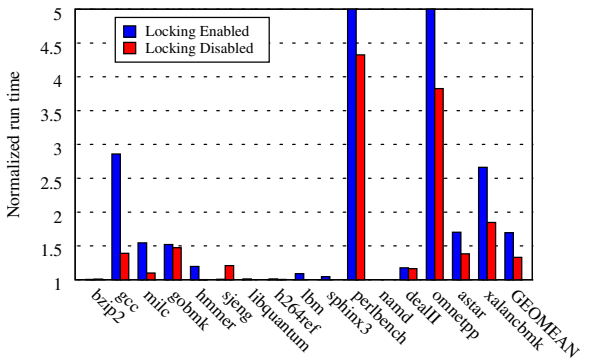


**Figure 1.** SPEC2006 performance overhead for FreeSentry scheme using METAlloc. Locking enabled and disabled represents normalized numbers with and without thread-safety, respectively.

SPEC2000 and SPEC2006. Moreover, our average number includes `omnetpp` run-time which has huge overhead. That is, `omnetpp` number further increases average value compared to FreeSentry. In conclusion, METAlloc seem to perform better than Label based system used in FreeSentry. Also, it has low memory overhead. However, introducing thread-safety dramatically increases performance overhead, making it impractical in production environment.

## 3. Overview

Large number of object allocations, deallocations and pointer propagations pose challenges in developing fast Use-after-Free detection system. Large scale applications like Web-Servers, Browsers are multi-threaded applications. Using dynamic analysis efficiently in multi-threaded application requires less thread synchronization. Recently proposed Use-after-Free detection systems introduce huge performance overhead or they have ignored thread-safety completely.

In this paper, we present and evaluate DangSan, a simple and efficient system to prevent Use-after-Free exploits during run-time. LLVM compiler instrumentation pass is used to insert run-time tracking function calls. We present optimal run-time data structure design for multi-threaded application. It removes a need for thread synchronization. We show that this lock-less design can be used practically in production servers that needs low overhead.

This paper has following contribution,

- We show that thread-safe Use-after-Free schemes in multi-threaded application can lead to huge run-time overhead. We implemented FreeSentry scheme using METAlloc. We evaluated this design with and with-out thread-safety.

- We propose DangSan, a novel, simple and efficient lockless system to efficiently store and retrieve pointer-object relationship in Multi-threaded applications.
- We implemented and evaluated DangSan for SPEC2006 CPU benchmarks and widely used Web Servers (`httpd`, `nginx` and, `lighttpd`).
- We verified DangSan correctness on recently discovered Use-after-Free (and Double-Free) vulnerabilities.

Section 4 discusses DangSan design. It discusses design assumptions, criteria and parameters. Performance and correctness evaluation is presented in Section 6. Related work is discussed in Section 8. Finally, Section 9 concludes our contribution.

## 4. DangSan Design

Data structure thread-safety in multi-threaded application introduces huge overhead. Efficient design should reduce concurrent access to data structure. In a simple design, we just need a list of pointers to the object. When object is freed, pointers are read from the list and invalidated. This simple design has following issues in multi-threaded environment. 1) Multiple threads writing to the same object list need to synchronized (i.e. object to metadata lookup need to be synchronized). 2) When pointer is no longer pointing to the same object, it should be removed from the list. Pointer to object metadata lookup is used to remove pointer information from the object metadata. Therefore, pointer to object metadata lookup need to be synchronized. 3) Same pointer can be inserted multiple times in the object list. DangNull and FreeSentry maintains pointer-object relationship. Relationship information does not change when pointer keeps pointing to the same earlier object. When pointer points to a new object, old relationship is removed and new relationship is inserted. Therefore, we need thread synchronization for object to metadata and pointer to metadata access.

Pointer remains in the old object list even after it no longer points to the object. This can lead to incorrect pointer invalidation when old object is freed. Pointer value can be checked to prevent incorrect pointer invalidation. Pointer is invalidated only when pointer points to any inbound object address. The same technique is needed when pointer is modified in non-instrumented code. We do not need pointer to object metadata. Also, removal of old pointer-object relationship adds extra performance overhead. Therefore, pointer to object lookup can be skipped, thereby removes need for a thread synchronization. Concurrent access to per object metadata (list) cannot be skipped. In a simple design, pointer list per object is required. Pointer list is equivalent to a log of pointers. Now, reducing concurrent access to pointer list per object boils down to a well-known problem of concurrent logs. Mostly, concurrent logging is on per-Thread basis. Concurrent logs are write efficient but with costly reads.

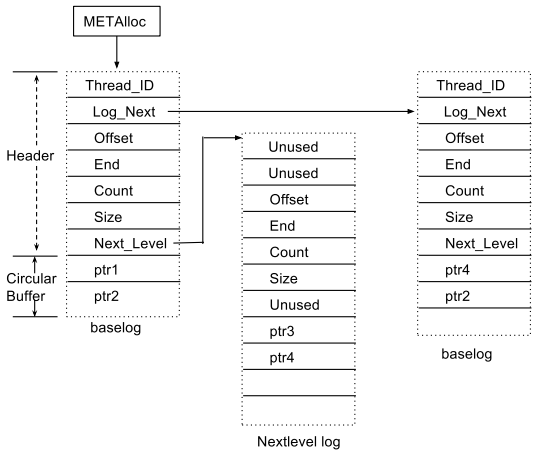


**Figure 2.** DangSan data structures for two threaded application.

Normally, number of pointer propagations are higher than the number of object allocations and deallocations. Therefore, pointer propagation tracking should be efficient. One approach is to skip pointer-to-object metadata lookup. That is no need to maintain old pointer-object relationship. This operation is equivalent to log-write. Motivated by the fact that log-write is efficient, we designed per-Thread per-object log data structure to track pointers pointing to the object in the log. Log read is required when object is freed. The design is similar to a log-structured file system [25] that maintains circular buffer to track I/Os. The design rationale behind DangSan is that the number of pointer registrations (write) are higher than the number of objects free (reads).

### 4.1 System Overview

Figure 2 shows DangSan design. Every object metadata has per-thread log. As we are not maintaining pointer-object relationship, only pointer address is needed and stored in the log. Object to metadata lookup retrieves corresponding log for a thread. Pointer tracking function retrieves corresponding log and writes pointer address into it. When object is freed, all per-Thread logs are retrieved. Pointer addresses are read from all the logs and invalidated. During invalidation, current value of pointer is read and checked to see whether it still points to the object.

**per-Thread Storage.** We need efficient retrieval of per-Thread log for a given object. One approach is to maintain object lookup table per-Thread. Similar to global shadow memory technique, it should also support range queries. Moreover, maintaining per-Thread object lookup table will incur huge memory overhead. Second approach is shown in Figure 2, where per-Thread log is maintained in global object list. Maintaining global object list (dynamically) needs thread synchronisation. Another approach is to maintain a

static array of log pointers per object. Thread specific *ID* can be used as index into the static array to retrieve corresponding log. Though this approach uses fast array indexing, it has high memory requirements. Second approach is memory efficient but needs concurrent access support. Log removal from dynamic object list is needed only during object free and not during pointer propagation or object allocation. Therefore, only concurrent insertions and reads are needed during pointer propagation. Lock-free insertions and reads can be achieved through compare and swap (*CAS*) atomic instruction [29]. Unique thread IDs can be used to identify thread log. One can use thread IDs assigned by the run-time system or maintain own thread IDs. Maintaining own thread IDs can help in reusing IDs. Reusing thread ID helps in reusing thread log. Dynamic object list can grow large when application has too many short lived threads. Thus, reusing thread log is necessary to reduce list traversal.

**Log Overflow.** In DangSan, pointer-object relationship is not maintained. That is, pointer addresses are not removed from old object logs. Depending upon the number of pointers to the object per thread, log overflow has to be handled. One approach is to reallocate log with larger size. But, reallocated log address can be different than the old log address. We need deletion operation to modify dynamic list with this new log address. That is, *CAS* atomic instruction for thread-safety can no longer be used easily. Second approach is to allocate new log for the thread and invalidate old log. Log invalidation is performed simply by setting thread ID maintained in the log to an invalid value. This way, thread can retrieve only a new log using thread ID. However, this approach increases length of dynamic list, thereby increases log retrieval time. Another approach is to introduce second level indirect log. Concurrent access is required for base level logs. Figure 2 shows Second level log. It is activated only when the base level log overflows. After base log overflows, each thread will find base log first and then second level log to store pointer address. Similar to base log, second level log can overflow. This can be handled easily by reallocating second level log with larger size. This operation does not require thread synchronization.

We have introduced few terms in our context $1)$ *Unique pointer:* Pointer address is stored only once in the log (i.e. Pointer has only one entry in the log). $2)$ *Duplicate pointer:* Pointer address is stored more than once in the log (i.e. Pointer has more than one entry in the log) $3)$ *Stale pointer:* A stored pointer in the log that no longer points to the object. $4)$ *Valid pointer:* A stored pointer in the log that points to the object. Choice of second level data structure also depends on the number of *Unique pointers* and *Duplicate pointers*. Hashtable can be used when *Duplicate* pointers are higher than the *Unique* pointers. But, HashTable introduces huge memory wastage.

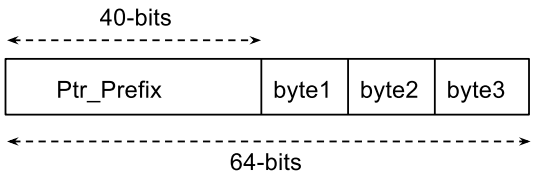


**Figure 3.** Pointer entry in the log, where Ptr_Prefix is a pointer value from 48-bits to 8-bits and byte is a lowermost byte of the pointer

**N-Lookbehind.** One of the reason for log overflow is large number of *Duplicate* pointers. One approach to remove duplicate pointers is to look-behind in the log for a pointer i.e. check pointer address against all stored pointers. Checking entire log for *Duplicate* pointers is a heavy operation. However, we can lookbehind *N* last offsets and skip pointer registration when pointer is within N-Lookbehind offset in the log. Normally, the same pointer is used to iterate over the object memory. This iteration occurs within a short period of time (e.g. in the loop). Depending on the value of $N$, we can eliminate large number of duplicate pointers. In normal scenario, *Unique* pointers are less than or equal to object size. When log size grows larger than object size, *Duplicate* and/or *Stale* pointers are higher than *Unique* pointers. Selection of *N* value is critical in removing *Duplicate* pointers but by considering performance overhead.

Moreover, *N*-Lookbehind can be used to increase log size utilization. On $64$-bit architecture, only lower $48$-bits represent user space virtual memory address. That is, upper most $16$-bits are not used. These two bytes can be used to store two more pointers i.e. (three pointers per log slot). Figure 3 shows a technique to make use of two bytes. We store all three pointers when $40$-bit prefix of all pointers is same. In other word, pointers differing only in lowermost byte can be stored together in the same log slot. Store $40$-bit pointer prefix in the uppermost $40$-bit of the log slot. Next, three remaining lowermost bytes (*byte1*, *byte2*, *byte3*) of the log slot are used to store lowermost byte of three pointers. Therefore, pointers within $256$ bytes range occupy the same slot in log. Here, *N*-Lookbehind is used to find a log slot where pointer prefix and a log slot prefix match. Following steps can be used to insert pointer into the log, $1)$ Find the empty slot by matching pointer prefix within *N*-Lookbehind entries. Matched slot is empty only when *byte2* or *byte3* is zero ($0x00$). $2)$ When empty slot is found, insert lowermost pointer byte at empty byte location. When pointer lowermost byte is $0x00$, swap this value with $byte1$. That is, pointer byte $0x00$ will always be stored at $byte1$ location. This is because, $0x00$ value denotes empty byte location. $3)$ Skip the registration when *Duplicate* pointer (i.e. both prefix and a byte matched) entry is found. $4)$ When no matching

log slot is found, insert 48 pointer bits into uppermost 48-bit of new log slot. Thus, *N*-Lookbehind strategy helps in comparing atmost $(N \times 3)$ pointers at the cost of *N* sequential memory access. In best case, log utilization increases by the factor of 3.

**Garbage Collection.** Another reason for log overflow is a large number of *Stale* pointers. Normally, with the time, old stored pointers tend to become *Stale*. One approach to remove *Stale* pointers is to treat log as a circular buffer. When log is about to overflow, iterate from log end to *Valid* pointer slot to find *Stale* pointers. Modify log end value with new *Valid* log slot. To determine pointer validity, read pointer value and check whether it still points to the object. For fast check, object bound information can be maintained in the object metadata. We need to grow the log when no *Stale* pointer slot is garbage collected. Note, when we store three pointers per slot, log slot is stale only when all three pointers are stale. Due to garbage collection, amount of work is shifted from object free context to pointer registration context. Total work remains the same with and without garbage collection. Garbage collection avoids log overflow, thereby prevents log reallocation cost.

**Pointer Liveness.** *Stale* pointers no longer point to the object. Object in which *Stale* pointer resides may no longer be live (i.e. unmapped). Accessing non-live pointer results into segmentation fault. We access *Stale* pointers in garbage collection and object free routines. To prevent invalid access to *Stale* pointers, one approach is to introduce new action for `SIGSEGV` signal in garbage collection and object free routines. Ignore `SIGSEGV` when signal is generated in these routines. Restore old `SIGSEGV` signal action at the end of above mentioned routines. Another problem with *Stale* pointers is that the pointer memory location might have been allocated to a new object. There exist a small window in object free routine between pointer value check and pointer invalidation operation. In this window, another application thread can write new value to the pointer which may get invalidated wrongly in object free routine. To avoid this problem, we use *CAS* atomic instruction to perform pointer invalidation only when pointer value is old.

### 4.2 Static Instrumentation

Run-time tracking function is instrumented statically using LLVM. Only pointer propagations are tracked (i.e. run-time tracking function is inserted before or after pointer assignment instruction). Allocations (`malloc`, `realloc`, `calloc`, `new`) and deallocations (`delete`, `free`) instructions have to be intercepted if these routines cannot be hooked during run-time. Listing 1 shows instrumented C code. Run-time tracking function `track_ptr()` is inserted after object allocation and pointer propagation code statement (Line 6 and 8). `track_ptr()` first retrieves metadata for an object and registers pointer address in the metadata. Similarly,

**Listing 1.** Static Instrumentation

```
int                                          1
main(int argc, char *argv[])                 2
{                                            3
  char *p, *q;                               4
  p = (char *)malloc(100);                   5
  track_ptr(&p, p);                          6
  q = p + 10;                                7
  track_ptr(&q, p + 10));                    8
  free(p);                                   9
  nullify_ptr(p);                            10
  return 0;                                  11
}                                            12
```

`nullify_ptr()` is inserted after object deallocation code statement (Line 10). It retrieves object metadata and invalidates all stored pointer by setting pointer to benign value NULL.

We are only interested in pointer assignments.

$$store\ rhs,\ lhs$$

We track only those store instructions where *rhs* is of pointer type. We pass both *rhs* and *lhs* as arguments to the run-time function. Stack objects are frequently created and destroyed. Benefit of tracking stack object compared to its performance overhead is very low. Similarly, global objects are destroyed (freed) only when application exit. Therefore, we conservatively filter out stores when *rhs* is a stack or global object. Moreover, incorrectly instrumented stack or global objects are skipped during run-time. Also, We skip store instrumentation when $rhs$ is a function pointer or a constant null pointer. When old pointer value is needed in tracking function, insert run-time tracking function before store instruction.

### 4.3 Parameters Selection

Performance of DangSan depends on the following three parameters. 1) **N-Lookbehind**: Increasing the value of *N* decreases the number of *Duplicate* pointers. It increases the chance of placing a pointer at already filled log slot. That is, it increases memory utilization of the log. However, increasing the value of *N* introduces cost of reading sequential memory. Distribution of pointer patterns in the log is application specific. Thus, choice of selecting value $N$ depends on the application. 2) **Log Size**: Increasing the log size decreases the number of log reallocations. However, it increases memory wastage. Selection of baselog size (default log size for each thread) and reallocation strategy is critical in maintaining balance between performance overhead and memory wastage. One reallocation strategy is to increase log size additively. This will reduce memory wastage but may need large number of reallocations. Another approach is to increase log size exponentially. This require logarithmic steps to reach maximum needed memory for an object. We

perform log resize operation only on the second level log. 3) **Thread-Specific operations**: Thread ID is maintained in thread-specific storage. Also, every log stores thread ID to identify thread-specific log. Maintaining and using thread-specific storage introduces extra memory access and computational checks. These extra performance overhead can be avoided for single threaded applications. One approach is to give compile time option to select the required variant (i.e. single threaded or multi-threaded). Another approach is to maintain thread ID in a register instead of thread-specific storage. Empirically for SPEC2006 benchmarks, $N = 3$ and baselog size $= 8$ (number of pointers) with exponential log resize gives better performance.

## 5. Implementation

We implemented DangSan on linux with METAlloc (metadata management) and `tcmalloc` [13] (custom memory allocator). Static instrumentation is built using LLVM 3.8. We do not instrument (malloc, calloc, realloc, new) and deallocation (free, delete) function calls. METAlloc provides a way to insert custom hook before and/or after allocation (deallocation) functions. We instrument only pointer store instruction. Run-time tracking functionality is provided in a static library.

**Application Compatibility.** We found that following cases can break application compatibility. 1) **Pointer subtraction**: Subtracting two dangling pointers pointing to the same object is a valid code statement. It can break due to DangSan. Our system sets pointer value to `NULL` when object is freed. Therefore, subtracting two dangling pointers result into 0 (i.e. incorrect value). To solve this problem, we invalidate pointer by setting the most significant bit of pointer value to 1. On Linux 64-bit, this invalid pointer value points into kernel address space. 2) **Off-by-One byte**: Some valid pointers can point out-of-bound by one byte. For example, STL vector has three fields, `start`, `next` and `end`. `start` points to start of the array memory. `next` points to next empty array location. `end` points to end of the array memory. `next` and `end` can point out-of-bound by one byte. Therefore, these pointers will get registered for wrong object. When wrong object is freed, `next` and `end` pointers will get invalidated. To solve this problem, we increase object allocation size by one. 3) **GCC issue**: We found one weird pointer usage in SPEC2006 `gcc`. `gcc` allocates a memory and stores allocated address minus some constant value into the pointer (i.e. no root address is stored). Registration of this pointer happens for wrong object. This pointer usage is wrong. Thus, we handled this issue as a special case during static instrumentation. For `GEP` LLVM instruction, when indices operand is negative constant or `SUB` LLVM instruction, we take root address of the object.

| Benchmarks | Normalized with Baseline | | Normalized with SafeStack | |
|---|---|---|---|---|
| | Heap Pointers | All Pointers | Heap Pointers | All Pointers |
| bzip2 | 1.01 | 1.01 | 1.01 | 0.99 |
| gcc | 1.42 | 1.59 | 1.33 | 1.48 |
| milc | 1.17 | 1.39 | 1.06 | 1.26 |
| gobmk | 1.27 | 1.29 | 1.28 | 1.30 |
| hmmer | 1.01 | 1.01 | 1.00 | 1.00 |
| sjeng | 1.41 | 1.42 | 1.33 | 1.34 |
| libquantum | 1.41 | 1.23 | 1.02 | 0.88 |
| h264ref | 1.02 | 1.02 | 1.00 | 1.01 |
| lbm | 1.11 | 1.08 | 1.03 | 1.00 |
| sphinx3 | 1.02 | 1.02 | 1.02 | 1.02 |
| perlbench | 4.49 | 4.87 | 4.05 | 4.40 |
| namd | 0.99 | 0.99 | 1.00 | 0.99 |
| dealII | 1.60 | 1.60 | 1.59 | 1.59 |
| omnetpp | 6.40 | 6.89 | 5.35 | 5.76 |
| astar | 2.21 | 2.24 | 2.14 | 2.17 |
| xalancbmk | 1.88 | 2.08 | 1.78 | 1.98 |
| GEOMEAN | 1.53 | 1.57 | 1.44 | 1.48 |

**Table 1.** SPEC2006 run-time overhead normalized with baseline and baseline-safestack. DangSan has 53% run-time overhead compared to baseline numbers when only heap pointers are tracked, whereas 43% compared to baseline-safestack. It has only 4% more degradation when all pointers (Stack, Heap and Global) are tracked compared to when only heap pointers are tracked.

## 6. Evaluation

We evaluated DangSan in terms of performance overhead and effectiveness. We used CPU-intensive SPEC2006 performance benchmarks. For baseline configuration, we compiled benchmarks with clang/LLVM 3.8.0. Moreover, we took numbers with one more baseline configuration (*baseline-safestack*) with SafeStack option enabled. METAlloc handles stack objects similar to SafeStack. However, DangSan do not track stack objects. Therefore, we compare DangSan with baseline-safestack to remove run-time overhead introduced by stack objects handling. Both baseline configurations use unmodified `tcmalloc 4.2.6` [13] as a custom memory allocator. For DangSan, we compiled applications with our extra LLVM transformation pass to instrument the code. We linked DangSan run-time library statically with the application. We used custom inliner LLVM pass to inline most of the our run-time tracking functions. For all configurations, compiler optimization is set to `-O3`. We ran benchmarks on 64-bit CentOS Linux with Intel Xeon CPU E5-2640 v3. We kept value of $N = 3$, baselog size $= 8$ (number of pointers) with exponential log resize strategy.

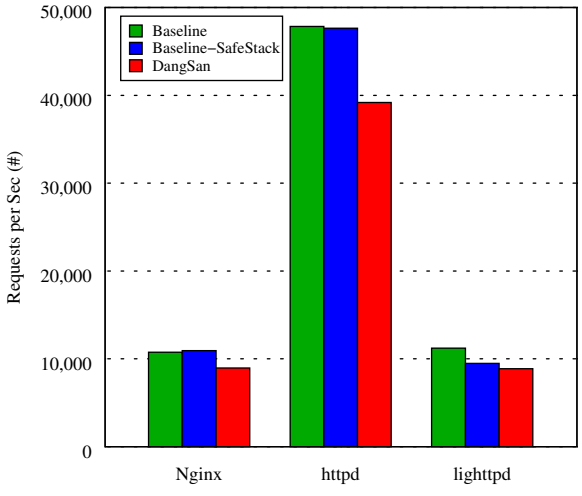


**Figure 4.** Web servers throughput degradation when compiled with DangSan. On an average throughput degradation is 12.8% (compared to baseline-safestack). Negligible degradation in service latency.

### 6.1 Performance Analysis

Table 1 depicts run-time overhead of SPEC2006 benchmarks. Benchmark run-times are normalized with baseline and baseline-safestack. Stack pointers are short lived. That is, attacker has a very short time to exploit vulnerability. Use-after-Free exploits using stack dangling pointers are very rare. Performance overhead of tracking stack pointers is very high compared to its benefit. Therefore, we performed experiments for tracking all pointers (Stack, Heap and Global) and only heap pointers. Our static instrumentation has conservative approach. We end up instrumenting every pointer store instruction. We skip registration for invalid pointers and objects during run-time. This reduces large number of false negatives. DangSan has average performance degradation (*geomean*) of 53% when only heap pointers are tracked. Moreover, it has only 4% more (compared to only heap pointers tracking) overhead when all pointers are tracked. SafeStack has low performance degradation of 0.1% [17]. Compared to baseline-safestack, DangSan has average performance degradation of 44% when only heap pointers are tracked. Similar to baseline, it has only 4% more (compared to only heap pointers tracking) overhead when all pointers are tracked. `omnetpp` and `perlbench` have high performance degradation of $6x$ and $4x$ times, respectively. State-of-the-art Use-after-Free detection schemes have not included either `omnetpp` or `perlbench` overhead. Therefore, we cannot directly compare our average performance overhead with other recent schemes. When `omnetpp` is excluded, average performance overhead is 42% (normalized with baseline) and 39% (normalized with baseline-safestack) when all pointers are tracked.

Moreover, we evaluated DangSan on widely used web servers like `nginx`, `httpd` and `lighttpd`. We measured service latency (runtime) and throughput (Number of requests per sec) degradation. To measure service latency, we downloaded 2GB file from server to client on localhost. We took average of 11 runs. DangSan has shown negligible service latency degradation. To measure throughput, we used `ApacheBench`. We triggered 25000 requests for static `index` page using 10 concurrent requests. Figure 4 shows throughput degradation compared to baseline and baseline-safestack. `lighttpd` has low throughput degradation of 6.5%, whereas `nginx` and `httpd` has moderate degradation of 18.2% and 17.7%, respectively. On an average (geomean), Web Servers have 12.8% throughput degradation compared to baseline-safestack.

Furthermore, we collected run-time statistics to understand performance degradation. For this, we instrumented DangSan run-time library functions. Table 2 depict runtime statistics for SPEC2006 benchmarks compiled with DangSan. Column 2 (Number of tracking calls inserted) represents a number of store instructions instrumented during static instrumentation phase. We have conservative static instrumentation pass. Therefore, we may end up instrumenting more than required. `gcc` and `xalancbmk` benchmark have almost $28K$ instrumented store instructions. Number of instrumented store instructions represent the number of pointer assignments found statically in the application. Column 3 (Pointer Registrations) denotes a number of times run-time pointer tracking function is called. `gcc` and `xalancbmk` have more number of instrumented stores than `perlbench` and `omnetpp`. However, the number of run-time pointer tracking function calls are less. This is because same instrumented tracking function is called many times. It can happen when we instrument frequently executing application functions or loops. This information can be used to perform DangSan optimizations. Column 6 represents number of allocated objects that are freed. Column 7 shows the number of times log overflow occurred. Column 4 shows percentage of pointer registrations that are for *Duplicate* pointers. For `lbm` benchmark, almost all pointer registrations are duplicates. For `sjeng` benchmark, pointer registrations are high with zero duplicates. It has only 5 allocated objects and no log overflow. It can happen when pointer registration is called for a large number of invalid objects (Stack or Global) or pointers. `perlbench` and `omnetpp` have large number of object allocations and deallocations. Therefore, performance overhead for these benchmarks could be because of object allocations/deallocations. We initialize object metadata in allocation function and invalidate pointers in deallocation function. However, `dealII` and `xalancbmk` benchmark have more number of object allocations and deallocations than `perlbench`. Thus, performance overhead in `perlbench` seems to come from a large number pointer tracking function calls. As discussed earlier, these calls are from frequently executed application

| Benchmarks | Number of tracking calls inserted | Pointer Registrations | Duplicate Pointers (%) | Objects Allocated | Objects Free (%) | Log Overflows |
|---|---|---|---|---|---|---|
| bzip2 | 30 | 766K | 50.5 | 28 | 85.7 | 0 |
| gcc | 28494 | 609915K | 81.7 | 1845K | 99.0 | 8829K |
| milc | 100 | 2329921K | 29.7 | 66K | 99.2 | 82320K |
| gobmk | 333 | 1837036K | 1.3 | 133K | 99.9 | 5931K |
| hmmer | 487 | 2403K | 22.1 | 1394K | 100.0 | 8K |
| sjeng | 14 | 271153K | 0.0 | 5 | 20.0 | 0 |
| libquantum | 19 | 130 | 49.2 | 149 | 100.0 | 0 |
| h264ref | 426 | 783K | 15.0 | 38K | 99.9 | 44K |
| lbm | 14 | 6K | 99.9 | 4 | 100.0 | 0 |
| sphinx3 | 433 | 312775K | 36.6 | 14225K | 98.5 | 16242K |
| perlbench | 8177 | 21572911K | 92.9 | 53673K | 96.8 | 113978K |
| namd | 60 | 61K | 46.6 | 1K | 99.8 | 0 |
| dealII | 6426 | 78545K | 40.4 | 151258K | 100.00 | 306K |
| omnetpp | 6907 | 16653803K | 10.8 | 267169K | 99.9 | 226082K |
| astar | 97 | 675195K | 6.5 | 3683K | 100.0 | 18641K |
| xalancbmk | 27839 | 2917543K | 36.5 | 135156K | 100.0 | 32602K |

**Table 2.** Run-time statistics for the SPEC2006 benchmarks

functions or loops. Therefore, advanced static instrumentation is required to avoid instrumentation for invalid store instructions.

**Pointer Patterns.** We do not remove pointer registration from old objects metadata. Therefore, object log has large number of *Duplicate* and *Stale* pointers. We studied pointer pattern distribution (the number of *Duplicate*, *Unique* and *Stale* pointers) in the log to fine-tune DangSan parameters.

Figure 5 shows pointer pattern distribution when a baselog overflow. We collected pointer pattern for first $1000$ objects that have overflown. The graph is plotted with increasing object size. Moreover, we collected statistics for benchmarks that incur huge overhead. Therefore, we chose two C benchmarks (`gcc` and `perlbench`) and two C++ benchmarks (`astar` and `xalancbmk`). We set baselog size to a very high value, $8K$ (number of pointers). Setting baselog size to a high value makes pattern clearly visible. We used *N*-lookbehind value as $4$. In Figure 5, red, green and blue area represents *Valid*, *Duplicate* and *Unique* pointers, respectively. All four benchmarks have large number of *Duplicate* pointers for any object size. Even after setting *N*-Lookbehind value to $4$, *Duplicate* pointers count is very high. Increasing value of *N* may decrease *Duplicate* pointers count but it has to trade-off with performance degradation. Furthermore, *Unique* pointers count increases with object size for all the four benchmarks. Therefore, default baselog size should be proportional to the object size. However, it may introduce huge memory overhead. Next, the sum of *Duplicate* and *Unique* pointers represent the total number of pointers in the log. We store at max three pointers per log slot. Therefore, the total number of pointers per log can be at max $8K\ (logsize) \times 3 = 24K$. For all the four benchmarks, on an average the total pointers count is above $11000$. In `gcc`, total pointers count reaches $23K$ for the object size $512$. That is, three times improvement in the memory utilization. Also, *Valid* pointers count is very low for all the four benchmarks. Note, *Valid* pointers count is inflated because pointers that are *Duplicate* and *Valid* are counted more than once. *Valid* pointers count is very low even after counting *Duplicate* and *Valid* pointers more than once. Therefore, we need garbage collection of *Stale* pointers to free the log space.

Furthermore, some pointers point to the same object but at different offset. These duplicate pointers may escape *N*-Lookbehind strategy. We introduced one more strategy to remove these duplicate pointers. We compare old pointer value to the new object address range. When pointer points to the same object, we skip pointer registration. This *Duplicate* pointers removal technique not only helps to remove *Duplicate* pointers within a thread but also across threads. This technique removed $20\%\ to\ 25\%$ *Duplicate* pointers for SPEC2006 benchmarks. For this technique, we need old pointer value in run-time tracking function. We slightly changed our static instrumentation pass. We insert run-time tracking function before store instruction instead of after store instruction. In run-time tracking function, we read old pointer value and check whether it still points to the same object. For fast check, we store object bound information in the object metadata. We store lowermost $32$-bits of object root address and object size together in the $64$-bit object metadata field. Therefore, METAlloc stores and retrieves $16$ bytes metadata, $8$ bytes for a pointer to a per-Thread log list

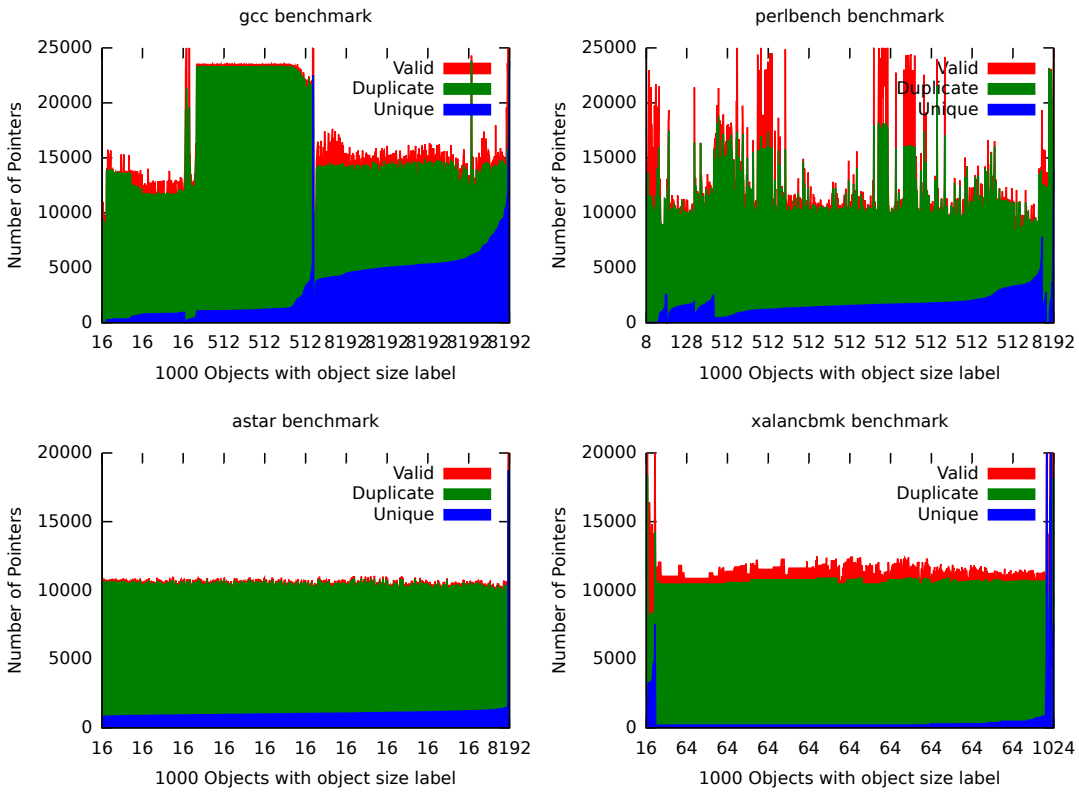


**Figure 5.** Pointer registration pattern for the benchmarks that incur huge overhead. First 1000 baselog overflow numbers are collected. N-Lookbehind value is 4. Size of the baselog is kept very high to clearly visualize pattern difference.

and 8 bytes for object information.

### 6.2 Correctness

We evaluated DangSan correctness on publicly available exploits. We chose following Use-after-Free (Double free) vulnerabilities.

**CVE-2010-2939** [4]: This is a double free vulnerability in OpenSSL client version OpenSSL1.0.0a and function `ssl3_get_key_exchange`. It is a highly critical vulnerability that results into denial of service or possible arbitrary code execution. We used exploit with the baseline configuration. It resulted into memory corruption error messages. Furthermore, we compiled OpenSSL1.0.0a with DangSan. We tried to exploit the compiled OpenSSL client. However, our system prevented double free. It aborted OpenSSL client due to invalid memory access.

```
src/tcmalloc.cc:290] Attempt to free invalid
pointer 0x80000000022ba510
./runclient: line 9: 20200 Aborted
```

Above message indicates that our system had invalidated pointer by setting uppermost bit to 1 when the object was freed first time.

### 6.3 Memory Overhead

Table 3 shows memory overhead on SPEC2006 benchmarks introduced by DangSan. `perlbench`, `omnetpp`, `xalancbmk` and `astar` benchmarks have huge memory overhead in gigabytes. Memory overhead also comes from the metadata management scheme, METAlloc. METAlloc stores and retrieves 16 bytes metadata for each object. It maintains metadata for all objects (including Stack and Global). We do not track Stack and Global objects. Thus, maintaining 16 bytes metadata for Stack and Global object can inflate memory overhead numbers. Some defences may not require stack or global metadata management. One of the improvement required in METAlloc is to have selective metadata management. We believe that METAlloc will be used for other defences along with DangSan. Another reason for the memory overhead is, we increase one byte for every object allocation to solve **off-by-one** byte application compatibility issue. However, `tcmalloc` allocates object with powers of 2 size. This tremendously increases memory overhead. One improvement to reduce memory overhead is to select additive increase log resize strategy. Table 3 also represents that increase in memory requirement increases performance overhead. This is directly proportional to the number of pointer propagations and object allocations.

| Benchmarks | Baseline RSS | DangSan RSS | Memory Overhead(MB) |
|---|---|---|---|
| bzip2 | 854.142 | 869.161 | 15.02 |
| gcc | 114.34 | 369.207 | 254.87 |
| milc | 647.039 | 843.874 | 196.84 |
| gobmk | 33.8847 | 257.16 | 223.28 |
| hmmer | 31.6067 | 61.5345 | 29.93 |
| sjeng | 179.271 | 202.317 | 23.05 |
| libquantum | 71.4484 | 88.8538 | 17.41 |
| h264ref | 34.2017 | 58.7903 | 24.59 |
| lbm | 412.537 | 428.296 | 15.76 |
| sphinx3 | 44.2548 | 617.769 | 573.52 |
| perlbench | 187.634 | 3012.63 | 2825.00 |
| namd | 50.1214 | 72.6506 | 22.53 |
| dealII | 196.355 | 1160.37 | 964.02 |
| omnetpp | 156.522 | 7626.64 | 7470.12 |
| astar | 87.5422 | 1993.17 | 1905.63 |
| xalancbmk | 318.452 | 2815.17 | 2496.72 |

**Table 3.** Memory overhead for the SPEC2006 benchmarks (MB)

## 7. Limitations and Future Work

DangSan tracks only heap objects. Stack objects can be supported similar to heap objects. METAlloc provides uniform way to store and retrieve metadata for stack objects. Static instrumentation has to be changed to insert run-time checks at the start and end of function calls. Moreover, we need to handle `longjmp` that performs non-local jumps. However, frequent stack object allocations and deallocations incur huge overhead. Furthermore, we conservatively instrument pointer store instructions. We can perform advanced inter-procedural and backward data flow analysis to eliminate stack and global objects accurately. One of the optimization is to avoid instrumentation for simple pointer arithmetic like *p++*. Similar to FreeSentry, we can move instrumented tracking function outside the loop if the same pointer is used in the loop. Also, we can provide function attribute to opt-out function from being instrumented.

## 8. Related Work

Much work has been done to prevent and detect use of dangling pointers. Mitigation scheme include techniques like static analysis and dynamic analysis. Other schemes propose customized memory allocators, memory error detection tools, safe languages etc.

**Static Analysis**: Static analysis performs source code or binary analysis to find memory errors statically. [12], SLAyer [8], needs inter-procedural pointer or data flow analysis. Static analysis does not cover all possible dangling pointer dereferences because object allocations, deallocations, pointer propagations and dangling pointer dereference can be in different modules, functions and threads. These techniques are not scalable for large applications. In many viable solutions, static analysis is combined with dynamic run-time check to detect Use-after-Free exploits efficiently .

**Dynamic Analysis**. Dynamic analysis tracks run-time pointer-object relationship. Recent schemes like DangNull [19], FreeSentry [30], UnDangle [9], CETS [20], Address Sanitizer [26] use run-time information to prevent Use-after-Free exploits. DangNull uses variant of red-black binary tree to efficiently store and retrieve memory object metadata. DangNull has huge average run-time overhead of $80\%$. Moreover, it does not track all pointers (Stack, Global and Heap). FreeSentry has low average run-time overhead of $25\%$. However, it has no support for multi-threaded applications. That is, it is unclear how much performance overhead FreeSentry incurs in production servers. UnDangle uses execution trace, taint tracking technique to identify memory locations associated for the same taint. It is useful in software testing. It needs full test coverage to identify all dangling pointers. It does not prevent dangling pointers use during program execution. Address Sanitizer detects memory errors during run-time. It extends compiler infrastructure LLVM to provide memory protection option. It covers most of the memory corruption bugs. However, it has on an average run-time overhead of $73\%$. [10] proposed improvements over Electric Fence [10]. It uses page protection mechanism to detect dangling pointer deference. It allocates new virtual page for every memory object. It has solved the virtual address space exhaustion problem by mapping multiple virtual pages to same physical page. However, this technique is inefficient for the applications that have large number of object allocations and deallocations.

**Memory Allocators**. Memory allocators designed to mitigate Use-after-Free vulnerabilities provide transparent solution. Cling [7] memory allocator is based on type-safe address reuse technique. Moreover, it does not use free memory for the metadata. Cling prevents type unsafe address reuse but it does not prevent unsafe dangling dereference for the same type object. DieHarder [24] memory allocator is a probabilistic approach to find memory errors. It randomize the location of heap objects that makes exploit hard to execute. DieHarder has low overhead but it is probabilistic (i.e. may not cover all dangling pointer dereferences).

**Memory error detectors**. Valgrind [23] and Purify [15] are widely used memory error debugging tools. Both the tools are used in software testing and debugging. Therefore, its effectiveness depends on the total test coverage. Moreover, Valgrind and Purify incurs huge performance overhead in the order of $10x$. GCCs Mudflap [11] performs dynamic memory access check by maintaining identifier for every al-

located object. Checking every memory access incurs huge performance overhead.

**Safe Languages**: Cyclone [16] is a safe dialect of C programming language. It prevents widely present spatial and temporal vulnerabilities of C language. It performs flow analysis and run-time checks. C applications require significant changes to port the application to Cyclone (as it has somewhat different syntax and semantics to simplify static analysis). It uses conservative garbage collection strategy which makes it slower than regular C language. CCured [22] adds memory safety to the C language by introducing run-time checks. It needs metadata for run-time checks. Porting of C application becomes difficult due to metadata requirement. Similar to Cyclone, CCured uses garbage collection that introduces significant run-time and memory overhead.

## 9. Conclusion

In this paper, we presented DangSan, a fast and efficient lock-less system to detect Use-after-Free exploits in multi-threaded applications. Thread-safety in multi-threaded program incurs prohibitively high run-time overhead. Our design (inspired from Log-structured file system) maintains per-Thread per-Object metadata that eliminates contention between threads. DangSan has moderate run-time overhead on CPU intensive benchmarks when all pointers (Heap, Stack and Global) are tracked. It has low throughput degradation for widely used WebServers. Our complete and efficient DangSan system can be used for large production applications.

## Acknowledgement

The author would like to thank his mentor and supervisor Erik van der Kouwe, Cristiano Giuffrida for their guidance and valuable assistance.

## References

[1] Cwe-416: Use after free. `https://cwe.mitre.org/data/definitions/416.html`. Accessed: 2016-09-18.

[2] Defence in depth:. Internet Explorer 0-Day: CVE-2010-3971. January 2011.

[3] National vulnerability database. NIST. `https://nvd.nist.gov/`. Accessed: 2016-09-18.

[4] Double free vulnerability in the. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2010-2939`. Accessed: 2016-09-18.

[5] Dangling Pointer: Smashing the pointer for fun and profit. `https://www.blackhat.com/presentations/bh-usa-07/Afek/Whitepaper/bh-usa-07-afek-WP.pdf`. Accessed: 2016-09-18.

[6] Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. `http://fort-knox.org/files/CSW2015.pdf`. Accessed: 2016-09-18.

[7] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers.

[8] J. Berdine, B. Cook, and S. Ishtiaq. Slayer: Memory safety for systems-level code. In *International Conference on Computer Aided Verification*, pages 178–183. Springer, 2011.

[9] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143. ACM, 2012.

[10] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 269–280. IEEE, 2006.

[11] F. C. Eigler. Mudflap: Pointer use checking for c/c+. In *GCC Developers Summit*, page 57. Citeseer, 2003.

[12] J. Feist, L. Mounier, and M.-L. Potet. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques*, 10(3):211–217, 2014.

[13] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, 2009.

[14] I. Haller, E. van der Kouwe, and C. Giuffrida. Metalloc: efficient and comprehensive metadata management for software security hardening. IEEE, 2008.

[15] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *In proc. of the winter 1992 usenix conference*. Citeseer, 1991.

[16] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[17] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, 2014.

[18] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[19] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Internet Society Symposium on Network and Distributed Systems Security*, 2015.

[20] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices*, volume 45, pages 31–40. ACM, 2010.

[21] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, pages 213–228. Springer, 2002.

[22] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[23] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[24] G. Novark and E. D. Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

[25] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *ACM SIGOPS Operating Systems Review*, volume 25, pages 1–15. ACM, 1991.

[26] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: a fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[27] F. J. Serna. The info leak era on software exploitation. *Black Hat USA*, 2012.

[28] H. S. E. Tutorial. Internet explorer use after free aurora vulnerability. *URL http://grey-corner. blogspot. com/2010/01/heap-spray-exploit-tutorial-internet. html*.

[29] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM, 1995.

[30] Y. Younan. Freesentry: Protecting against use-after-free vulnerabilities due to dangling pointers. In *Proceedings of the 2015 Internet Society Symposium on Network and Distributed Systems Security*, 2015.

[31] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. Paricheck: an efficient pointer arithmetic checker for c programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 145–156. ACM, 2010.