DangSan: Scalable Use-after-free Detection

Erik van der Kouwe
Vrije Universiteit Amsterdam
vdkouwe@cs.vu.nl

Vinod Nigade
Vrije Universiteit Amsterdam
v.v.nigade@student.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

Abstract

Use-after-free vulnerabilities due to dangling pointers are an important and growing threat to systems security. While various solutions exist to address this problem, none of them is sufficiently practical for real-world adoption. Some can be bypassed by attackers, others cannot support complex multithreaded applications prone to dangling pointers, and the remainder have prohibitively high overhead. One major source of overhead is the need to synchronize threads on every pointer write due to pointer tracking.

In this paper, we present DangSan, a use-after-free detection system that scales efficiently to large numbers of pointer writes as well as to many concurrent threads. To significantly reduce the overhead of existing solutions, we observe that pointer tracking is write-intensive but requires very few reads. Moreover, there is no need for strong consistency guarantees as inconsistencies can be reconciled at read (i.e., object deallocation) time. Building on these intuitions, DangSan's design mimics that of log-structured file systems, which are ideally suited for similar workloads. Our results show that DangSan can run heavily multithreaded applications, while introducing only half the overhead of previous multithreaded use-after-free detectors.

CCS Concepts • **Security and privacy → Systems security**

Keywords Dangling pointers, use-after-free, LLVM

1. Introduction

Use-after-free vulnerabilities caused by inadvertent use of dangling pointers are a major threat to systems security. Compared to other types of vulnerabilities, use-after-frees are relatively hard to detect by either manual inspection or static analysis, since there may be a long time between storing a pointer to memory, terminating the lifetime of the object it points to, and dereferencing the stored pointer resulting in undefined behavior. While setting pointers to `NULL` after freeing the object that they point into would seem to be a relatively simple solution, in practice this is hard to do because copies of pointers can be spread throughout the many data structures of the program. Moreover, dangling pointers that result in later use-after-free vulnerabilities are readily exploitable [50] and often the exploitation method of choice in real-world attacks [37, 51].

There are various ways to shield software from dangling pointer exploits. A successful exploit requires three elements: (1) a pointer pointing into an object that has been deallocated, (2) another memory object relevant to the attacker reusing the memory area that used to be occupied by the deallocated object, and (3) a (use-after-free) instruction where this pointer is dereferenced (read from or written to). Consequently, there are three main ways to defend against use-after-free vulnerabilities, each removing one of the elements of a successful exploit. Secure memory allocators [12, 14, 40, 45] prevent exploitation of use-after-free vulnerabilities by avoiding allocation of objects reusing memory originally owned by deallocated objects. Unfortunately, these approaches do not fully defend against deliberate attacks because they can be circumvented through heap spraying or massaging [37]. Approaches that prevent dangling pointers from being dereferenced [19, 39] avoid this problem by keeping track of each pointer's origin and verifying whether that instance of the pointer has previously been deallocated. However, the tracking and the need to instrument every pointer dereference to perform a check cause these approaches to incur very high overhead, hindering practical adoption.

The most promising design for practical run-time use-after-free defenses prevents the creation of dangling pointers altogether. This can be achieved by keeping track of pointers to each memory object and replacing them with invalid pointers once the object is freed. Unlike the original dangling pointer, the invalid pointer cannot be used to leak or corrupt memory as any attempt to dereference it will be detected as a segmentation fault. To the best of our knowledge, there are currently two systems that follow this approach: FreeSentry [51] and DangNULL [37]. While FreeSentry offers reasonable overhead, much of its performance benefits

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

ISBN 978-1-4503-4938-3/17/04. . . $15.00

DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/3064176.3064211

derive from the inability to support multithreaded programs, hindering again practical adoption. DangNULL does support multithreading, but imposes prohibitive run-time overhead. Moreover, it is particularly prone to false negatives, as it only supports tracking pointers stored on the heap. For these reasons, we conclude that there is currently no practical and widely applicable detection system available against use-after-free vulnerabilities.

In this paper, we present DangSan, a new use-after-free detection system which is efficient and readily applicable to real-world C/C++ programs. Our overhead is on par with the most efficient (and thread-unsafe) pointer tracking solution, while offering much better performance, scalability, and detection coverage than alternatives that do support multithreaded programs.

To efficiently support thread-safe semantics, we propose a new lock-free design inspired by the way log-structured file systems operate [44]. The key insight is that, much like log-structure file systems, our system needs to perform many frequent writes (due to pointer tracking instrumentation at every pointer store instruction) but relatively infrequent reads (due to pointer invalidation instrumentation at every `free` call). Moreover, using locks to guarantee consistent access to shared data structures at every write is not only inefficient, but also unnecessary. Since we can verify at free time whether a pointer to the freed object is still stored at its previously recorded location, we can efficiently recover from stale and duplicate pointer entries in our records. Based on these insights, our design refrains from using complicated shared data structures and simply opts for append-only per-thread logs for each object in the common case.

While our new design offers competitive run-time performance and scalability, the redundancy and stale pointers in the per-thread logs can cause significant memory overhead. To address this problem, we present a number of optimizations to reduce this overhead while retaining the good performance of the basic design. We use a fixed-length lookback to reduce redundancy and use a hash table as a fallback for excessively large logs. This fallback avoids the risk of unbounded memory consumption in case duplicate pointers occur with cycles longer than the lookback. In addition, we use static analysis to reduce the number of pointers that need to be tracked and pointer compression to store multiple pointers in a single log entry. Using these techniques, we show how to achieve a reasonable balance between run-time performance and memory overhead.

Contributions

- We present a novel design for practical use-after-free detection inspired by log-structured file systems. Our design combined with an efficient shadow memory-based metadata management scheme can offer considerably better performance and scalability than existing multithreaded dangling pointer detectors.
- We present DangSan, a prototype implementation of our design based on the LLVM compiler framework [36] and the scalable tcmalloc allocator [24]. To foster further research in the field, we will open source our DangSan prototype upon acceptance.
- We present an evaluation of our prototype, demonstrating that DangSan offers a new efficient and scalable design point in use-after-free detection, at the cost of moderate but realistic memory overhead in real-world settings.

Roadmap The remainder of this paper is laid out as follows. Section 2 presents the threat model addressed by DangSan. Section 3 give an overview of the high-level structure of DangSan and Section 4 introduces the individual components in more depth. Section 5 discusses implementation details for our DangSan prototype and Section 6 presents our optimizations. Section 7 discusses some limitations of our design and their impact on the practical use of DangSan. Section 8 presents and discussed our experimental results. Section 9 compares DangSan against the state of the art in use-after-free defenses. Finally, Section 10 concludes the paper.

2. Threat model

We assume that the vulnerable program contains one or more use-after-free vulnerabilities, where an attacker can force the program to read from or write to a dangling pointer to a deallocated memory object that potentially overlaps with one or more other memory objects. We assume that defenses are already in place for other types of memory errors such that they cannot be exploited. Specifically, we assume that the attacker cannot corrupt our data structures, which can be achieved through orthogonal defenses such as hardened information hiding [26, 41], re-randomization [20, 25, 49], or efficient isolation techniques [22, 35, 48]. Such solutions can provide metadata integrity with low performance impact (typically less than 5% on average on SPEC CPU2006).

3. Overview

We designed DangSan as a LLVM compiler plug-in [36] combined with a library to interpose on memory allocator operations. To use DangSan, users need to pass the appropriate compiler flags when compiling their application to protect it against use-after-free exploits. Our framework automatically tracks per-object pointers and invalidates them when the corresponding object is freed.

Figure 1 provides an overview of DangSan's components, with the white parts already present in an unprotected program and the gray parts added by DangSan. To be able to invalidate pointers at free time, we need to track the memory locations containing pointers that point to each object in memory. The *pointer tracker* is a compiler pass that instruments all the stores of pointer-typed values. Whenever the program stores a pointer to memory, the pointer tracker in-

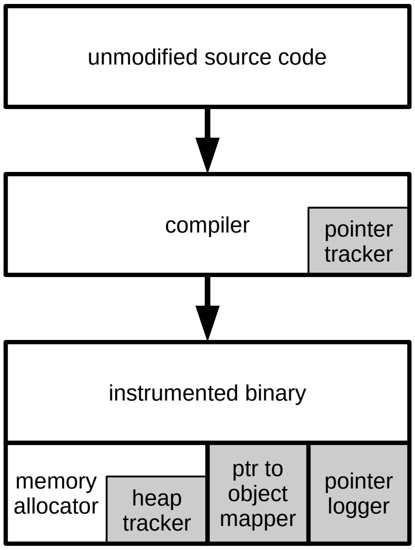

Figure 1. Overview of DangSan

strumentation invokes the *pointer-to-object mapper* to look up which memory object the stored pointer points into. We use the term *memory object* to refer to a contiguous region of virtual memory allocated by a single call to `malloc`, `realloc`, etc. Finally, the pointer tracker calls the *pointer logger* to register the newly stored pointer. The pointer logger associates each memory object with a per-thread log that stores a list of memory addresses containing pointers into that object. The *heap tracker* uses this information to be able to invalidate any references to a memory object once the object is freed. This component hooks into the memory allocator to get notified whenever a memory object is allocated or deallocated. We describe the individual components in more details in Section 4.

4. Design

In this section, we discuss the design of DangSan. Each subsection describes one of the components in Figure 1 and its interactions with the other components. This section discusses the key elements of our design, while optional elements to improve performance are discussed in Section 6.

4.1 Pointer tracker

The main challenge when invalidating dangling pointers to detect uses-after-free is to locate all the pointers to freed objects. For this purpose, DangSan relies on a pointer tracking compiler pass. Our pointer tracker scans program code to identify all the store instructions with pointer types and, for each of them, inserts a call to a function `registerptr` supplied by a static library. This function receives two parameters: the store location (a pointer to a pointer) and the stored value (possibly a pointer to a memory object). The function first uses the `ptr2obj` functionality of the pointer-to-object mapper (see Section 4.3) to look up the memory object referenced by the stored pointer value. Next, it uses the `logptr` functionality of the pointer logger (see Section 4.4) to associate the store location with the object. Figure 2 shows an overview of the interactions of the pointer tracker with the rest of DangSan. This approach allows us to build a list of all the memory locations containing pointers into each allocated object.

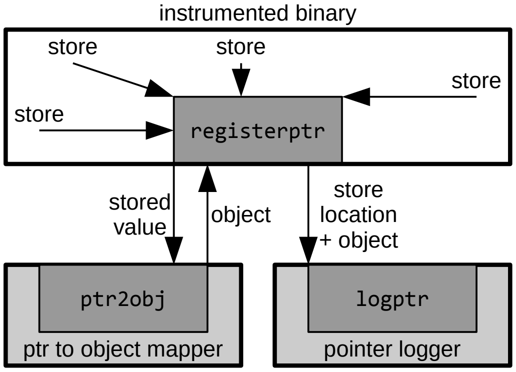

Figure 2. Interactions of the pointer tracker

4.2 Heap tracker

In addition to tracking pointers to memory objects, DangSan also needs to track the memory objects themselves. In particular, it needs to initialize the pointer-to-object mapping whenever a new object is allocated and invalidate all pointers to an object whenever it is freed. For this purpose, DangSan hooks into allocation/deallocation calls (e.g., `malloc`, `realloc`, `free`) in the memory allocator. In this section, we discuss how DangSan handles each of these calls.

Whenever the protected program uses the `malloc` (or similar) call to allocate memory, DangSan needs to ensure that every pointer pointing into the new object can be mapped to the new object. For this purpose, DangSan uses the `createobj` interface of the pointer-to-object mapper. Figure 3 shows the interactions within DangSan whenever memory is allocated.

Whenever the heap tracker intercepts a `free` call, DangSan performs its main job: it invalidates any pointers to the memory object about to be freed. This action is performed by the pointer logger, which already holds all the necessary information. The heap logger retrieves the object identified by the freed pointer using the pointer-to-object mapper and passes it along to the `invalptrs` functionality of the pointer logger. Figure 4 shows interactions within DangSan whenever memory is deallocated.

When the program calls `realloc`, we need to distinguish three possible cases: the object remains as is, the object is resized in place, or a new object is allocated and the contents of the old object are copied into it. In the first two cases, the

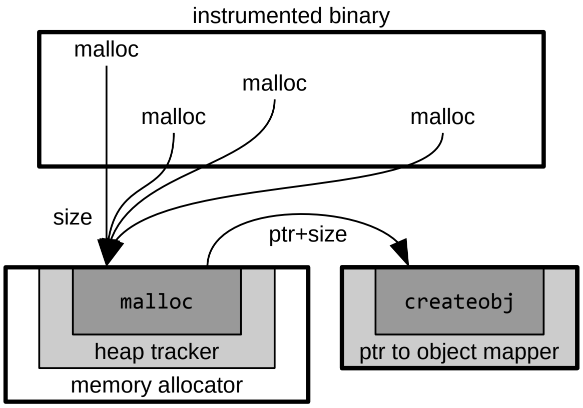

Figure 3. Interactions of the heap tracker: `malloc` call

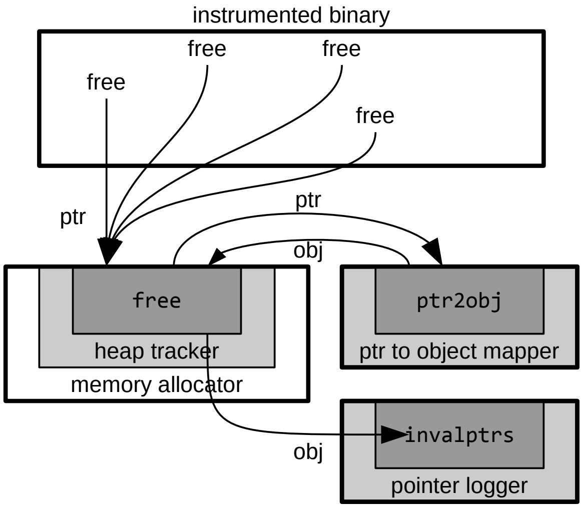

Figure 4. Interactions of the heap tracker: `free` call

pointers to the object remain the same so they need not be invalidated. However, if a block is grown in place we do need to extend its mapping in the pointer-to-object mapper. This is done by simply using the `createobj` interface again, which overwrites the old mapping. In the third case, the hooked `malloc` and `free` calls ensure that the object mapping is adjusted and pointers to the old object are invalidated.

4.3 Pointer-to-object mapper

To be able to associate pointers with metadata for the memory objects, we need to keep track of a mapping from pointers to objects, associating metadata for use by the pointer logger with each object. We use this mapping to implement the *ptr2obj* call used by the pointer tracker. The pointer-to-object mapper adds new mappings whenever the heap tracker detects that a new memory object has been allocated and invokes the *createobj* interface. In this section, we describe the data structures used to perform these tasks efficiently.

The data structures need to support efficient range queries, as we want to invalidate every pointer pointing within the memory range of an object when that object is freed, not just those pointing to the start of the object. This means that we cannot use hash tables, which do not allow for range queries. Lookups need to be very efficient to be able to achieve low overhead because they are performed every time a pointer is written to memory. This means that trees are not a suitable data structure either, as their lookup performance degrades as more memory objects are allocated [37]. For this reason, we have opted to use memory shadowing [45], which provides efficient constant-time range queries.

The intuition behind memory shadowing is that, if every memory object is aligned to a multiple of n bytes, we can divide a memory address by n to obtain an index such that bytes with the same index are always inside the same memory object. We then allocate a huge array of metadata and use the index to look up the metadata for a pointer. In this design a memory object may span multiple entries in the metadata array, in which case the metadata must be duplicated across all entries. One important variable is the ratio between the amount of program data and the amount of metadata, which we will refer to as the *compression ratio*. If we assume the guaranteed alignment n is the same everywhere in memory and each metadata entry is m bytes, the optimal compression ratio is $\frac{n}{m}$. This approach is used in traditional memory shadowing systems, such as the one used by AddressSanitizer [45]. Such systems can look up metadata using just a single memory read. One problem with traditional memory shadowing, however, is the fact that the allocation of large memory objects requires a proportionally large area of shadow memory to be initialized. While this can be improved by increasing the compression ratio, doing so requires that objects to be aligned to larger boundaries to ensure that no two objects are mapped to the same location in shadow memory. This increases memory fragmentation, which is undesirable for both performance and memory usage. The alternative is to reduce the amount of metadata (like in Baggy Bound Checking [13], which uses just one byte of metadata per object). However, DangSan requires a full pointer (eight bytes on 64-bit systems) to be able to map memory objects to pointer logs. As a consequence, constant compression ratio approaches incur unacceptable overhead for pointer tracking.

To be able to efficiently map pointers to pointer logs, we have chosen to use a variable compression ratio memory shadowing scheme proposed in recent research [28, 29]. With the variable compression ratio approach, the mapping from memory addresses to shadow memory is not the same everywhere. Instead, we keep track of the alignment of memory blocks and use a higher compression ratio for objects aligned to larger boundaries. We use two levels of shadow

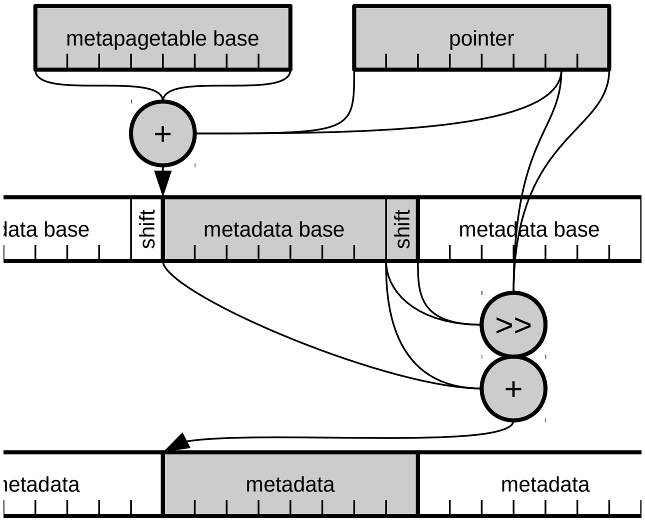

Figure 5. Using the metapagetable to find metadata for a pointer

memory and, as a consequence, metadata lookups require two memory reads. Figure 5 shows how DangSan uses this approach to look up metadata for a pointer. As a first step, DangSan operates a fixed compression ratio lookup to determine the variable compression ratio for that particular pointer. This shadow memory region is a form of *metapagetable*, which stores a single eight-byte `long` value for every 4096-byte memory page. Seven bytes specify a pointer to an array of metadata for memory objects within the memory page, while the eighth byte specifies the compression ratio to apply. Finally, we locate the metadata by computing the offset of the original pointer in the memory page, shifting it using the variable compression ratio and adding the base metadata pointer for the page.

4.4 Pointer logger

The pointer logger is the most important component and the key enabler of DangSan's practical use-after-free detection strategy. It keeps track of the pointer locations reported through the `regptr` interface and invalidates pointers to an object when `invalptrs` is called.

The core insight that drives the design of our pointer logger is the fact that the workload is write-intensive (any store of a pointer type invokes `regptr`) while there are relatively few reads (`invalptrs` is only invoked for `free` calls). Moreover, there is no need to enforce consistency between threads. Duplicate logging of the same pointer location is not harmful and there is no need to remove pointer locations that no longer point to an object because we can verify this at free time. We draw inspiration from log-structured file systems [44], which are optimized for similar workloads: instead of using complicated data structures requiring locks to ensure consistency, in the common case we simply keep a per-thread log for each memory object to which we merely add new pointer locations. This strategy yields a lock-free design which completes frequent (and instrumented) write operations very quickly.

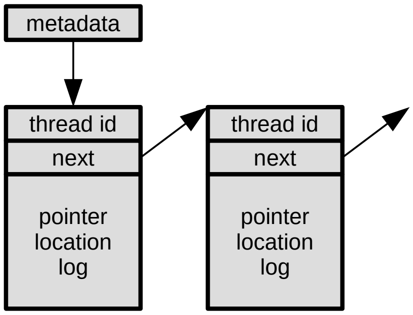

Figure 6. Pointer logger's main data structure

Figure 6 shows the core data structures of the pointer logger. Any request to the pointer logger for a specific object includes metadata that identifies the object, which the caller can look up from a pointer using the pointer-to-object mapper. This metadata points to the first entry in a lock-free singly linked list of logs, one for each thread that has stored pointers to the object. The pointer logger walks the list until it finds the log for the current thread. If there is no log for the current thread, it allocates a new log and adds it at the end of the linked list. We use a compare-and-exchange instruction to perform this operation in a thread-safe fashion. Since modifications to the list are rare, this design ensures few compare-and-exchange conflicts and, as a result, good parallelism and scalability in practice.

While the choice to use a singly linked list even in programs with many threads seems counterintuitive given its worst-case behavior, the latter occurs only if many threads propagate pointers to the same object. Since in practice most memory objects are either thread-local or shared among few threads [33], in practice operations on this list complete very quickly even with many running threads. A design alternative would be to preallocate an array containing an entry for every thread. This would allow $O(1)$ lookups, but would also incur high memory overhead because many entries would be unused for most objects. Moreover, it would require the programmer to set an upper bound on the number of threads that can exist simultaneously.

The pointer logger provides the `regptr` interface to add a pointer by specifying the object's metadata and the location where the pointer is stored. In the common case, it uses the log entries embedded in the pointer location log shown in Figure 7. It first verifies whether the same pointer location has been added recently by looking back a fixed (but compile-time configurable) number of entries and, if it is not found, adds the pointer location at the end of the log. The look-back prevents excessive log growth in cases where a pointer to the object is stored into the same location over and over again (e.g., loops with a pointer iterator variable).

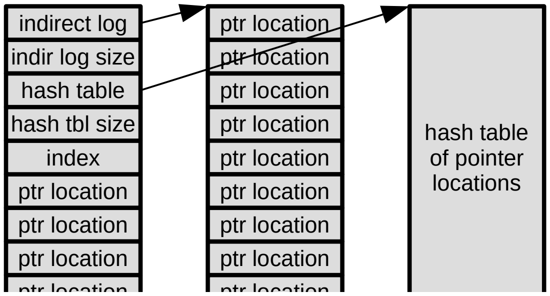

Figure 7. Pointer location log

In our experiments, we have chosen to use a lookback size of four. While the optimal lookback size differs slightly between benchmarks, overall performance is generally similar in the range between one and four, and begins to degrade with higher numbers. We opted for four so as to save memory at near-optimal performance. Performance could be improved somewhat by selecting the ideal size for each individual benchmark, but preliminary measurements suggest the potential performance gain is not large enough to justify the additional complexity.

If the embedded log is full, we use a hash table rather than a log. The reasoning is that we do not want the pointer log to grow to arbitrary sizes if there are duplicate pointers that our look-back mechanism cannot eliminate. This way, our design yields the good performance offered by a simple log for the common case while preventing excessive memory usage on pathological cases.

Whenever the program frees a memory object, the heap tracker invokes the `invalptrs` functionality to invalidate any pointers into the to-be-freed object. This function walks through the log and, if allocated, the hash table to retrieve all locations where pointers into the object may be stored. For each of them, we first verify whether it still points into the object. If a location contains a valid pointer value, we invalidate it using a compare-and-exchange operation. This scheme prevents a race condition where we mistakenly overwrite a new pointer written by another thread to invalidate the old one.

There are two ways in which our check to determine whether the pointer still points into the object could be unsafe: an out-of-bounds pointer to another object is mistaken for a pointer to the freed object and an integer stored at the location where the pointer used to be happens to have the same value as the pointer. We avoid the former case by increasing all `malloc` allocation sizes by one byte, guaranteeing that a pointer to the end of an object does not point to another object. Pointers that are further out of bounds result in undefined behavior according to the C standard [3], so that, even in the very unlikely case of accidental invalidation, we would merely catch an additional bug.

For the latter case, values with a non-pointer type that still take the value of a pointer within the object would be incorrectly invalidated if they end up in a place that previously stored a pointer to the object, but this case is irrelevant in practice. On modern 64-bit systems, integer values are unlikely to match *any* valid pointer value in the address space, enabling type-accurate conservative garbage collection [32]. The likelihood of a match decreases even further for DangSan, where we need to limit the analysis to only pointers in the target object.

A more interesting case is partial type-unsafe memory reuse [18]. For example, this may occur when a memory location containing a pointer value is reused by another object to store a single byte (e.g., `char`) value, overwriting only its least significant byte after reuse. The end result is a pointer-sized value that may still match a valid address in the range of the original pointed object. To address this problem, we only overwrite the most significant byte when invalidating pointer values at free time (see below). This leaves the unlikely case of code reusing a pointer value as a pointer-sized integer value and only initializing its most significant byte (while leaving other bytes uninitialized). Although these hypothetical instances can be further mitigated with a secure deallocation strategy [21], we do not believe them to be of concern in practice. We also note that these cases are not explicitly addressed by existing solutions [51].

Another potential issue is the case where the object storing the pointer has been deallocated and its memory has been returned to the operating system. In this case, our attempt to read the pointer from the stored pointer location would result in a segmentation fault. For this scenario, we catch the `SIGSEGV` signal and skip pointer locations that trigger this fault (alternatively, we could use Intel TSX to catch the fault directly in userland [34]). While this could be prevented by removing pointers from the log whenever the memory they are stored in is released, doing so efficiently would require an additional data structure to keep track of pointers based on the object they are stored in rather than just the one they point to. Keeping this data structure up-to-date would require considerably more work on pointer propagation, resulting in more overhead. Through these safeguards, we can implement `invalptrs` in a way that does not result in any false positives in practice without the need for expensive mechanisms to track pointer deletions.

While we have discussed how to find pointers to invalidate, we have not yet determined which value to overwrite the original pointer with. Some prior work writes a fixed invalid pointer value (DangNULL [37]) while others modify the pointer by setting bits that are not allowed in user-space pointers (FreeSentry [51]). We opt for the latter approach, which has the benefit of reporting a fault address that can still be related to the original pointer if the dangling pointer is dereferenced, making debugging easier. Moreover, some programs such as `soplex` rebase pointers after a `realloc` call by adding the difference between the old and the new pointer. This approach still results in correct behavior if an unused bit of the old pointer is set. Finally, this approach efficiently mitigates the issues with partial type-unsafe memory reuse detailed earlier. As such, we believe setting the most significant bit (resulting in an invalid address in canonical x86-64 form) is the most practical as well as prudent approach.

5. Implementation

We have implemented a prototype of our DangSan design to protect C and C++ programs running on 64-bit Linux. We implemented the pointer tracker as a compiler pass in the LLVM compiler framework [36]. We use link-time optimizations (LTO) and invoke our pass through the LLVM-gold plug-in for the GNU gold linker to run on the LLVM bitcode of the full program. This allows our system to be easily integrated into common build systems for C and C++ programs by simply passing compiler flags to enable LTO, using the GNU gold linker instead of the traditional BFD linker, and specifying a linker flag to invoke the DangSan pointer tracker pass. Note that our design as such does not fundamentally require LTO and could be implemented without it. We used LTO to simplify the implementation and integration in existing projects.

The pointer-to-object mapper is based on the tcmalloc [24] memory allocator, which enforces a memory layout that allows for efficient variable compression ratio memory shadowing—as also observed in prior work [28, 29]. As a consequence, we implemented our heap tracker as a tcmalloc extension. Finally, we implemented the pointer logger as a static library linked against the final binary whenever DangSan is enabled through a linker flag.

6. Optimizations

To achieve good performance and reasonable memory overhead, we have implemented a number of optimizations in our prototype compared to the design described in Section 4.

Even though we have designed DangSan to reduce overhead on instrumented pointer stores, this operation is very common and therefore still has an important impact on performance. Furthermore, a large number of instrumented pointer stores can result in many duplicated entries in DangSan's pointer logs, which may lead to excessive memory overhead and, as a side effect, performance overhead (e.g., due to poorer cache utilization).

To further speed up DangSan and lower its impact on the memory footprint, we use static analysis to reduce the number of required instrumentation hooks. We can eliminate calls to `regptr` in two cases: loops and pointers we know are already registered for a given object. In the case of loops, we use an optimization inspired by prior work [51]: if conservative static analysis shows that the loop body does not call `free` (possibly through other functions), it is possible to move loop-invariant pointer registrations outside the loop. This allows us to eliminate pointer registrations in locations that are overwritten before `free` is called.

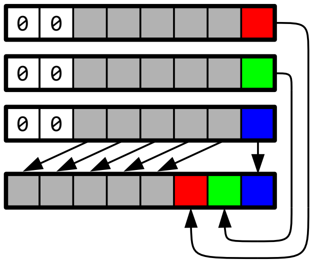

Figure 8. Pointer compression

As for pointers that we know are already registered for a given object, we specifically consider pointers derived from pointer arithmetic. This is justified by the fact that the C standard considers pointers that go out of bounds to result in undefined behavior except for the special case of pointing directly past the last element of an array [3]. Since we cover this case by increasing all allocation sizes by one byte, we know that pointer arithmetic is never allowed to cause a pointer to point to a different memory object. As a consequence, we can safely omit pointer registrations in cases where the program merely adds to or subtracts from a stored pointer. Since we register only the address of the pointer and not its value, pointer arithmetic requires no updates in the metadata. Both these optimizations reduce the number of pointer registrations, improving run-time performance and memory usage.

If a program does not conform to the standard and computes intermediate pointers that are too far out-of-bounds, it is possible that these pointers are registered as pointing to another object. If that object is freed, the pointer would be invalidated, resulting in a false positive. This could result in a segmentation fault if the invalidated pointer is dereferenced, revealing that the program performs nonconforming pointer arithmetic and should thereby be fixed to ensure adherence to the standard.

The pointer logs make up most of the memory overhead imposed by DangSan. To reduce this overhead, it is important to store the pointers as efficiently as possible. We use the fact that on 64-bit x86 CPUs, the two most significant bytes in pointers are unused for user-space pointers. If we need to store up to three pointers that only differ in their least significant byte, we shift the part they have in common two bytes to the left to eliminate the zeroes and then use the three least significant bytes to store the least significant bytes of the three pointers. Figure 8 shows this approach to compress pointers. This optimization can save up to a factor three on space overhead depending on spatial and temporal locality in pointer stores.

7. Limitations

While our design should be able to efficiently detect dangling pointer vulnerabilities in practical settings, it has a few fundamental limitations. We discuss these limitations in this section.

Our design requires source code to be available because we infer which stores are pointer-typed using information from the compiler's intermediate representation and insert our pointer tracker instrumentation at that same level. This information is lost after the compiler generates binary code. While it would be possible to build a similar system that uses only the binary code, we believe this would not result in a practical system. There are two options to implement such a system: using a static approach or a dynamic approach. A static solution would perform static analysis on the binary code to identify pointer-typed values and then rewrite the binary to instrument stores of these values. However, static analysis is generally unable to identify many cases where pointers are copied without being dereferenced or passed to library functions with pointer-typed arguments [47]. In other words, this approach would likely result in low detection coverage in practice. A dynamic solution could effectively track pointers using techniques such as taint tracking, which, despite efforts, still imposes significant run-time overhead [17]. Moreover, every store would have to be instrumented if it is not known beforehand whether the stored value will be of a pointer type, further increasing run-time overhead. For this reason, we believe requiring source code to be available is a practical (rather than fundamental) limitation.

A related limitation is the fact that we cannot track metadata for uninstrumented shared libraries. This means that pointers stored in those libraries will not be invalidated when the objects they point to are freed (reducing use-after-free detection coverage). This limitation is easy to overcome if the source code of the libraries is available. One can simply recompile those libraries using the DangSan compiler options.

In some particular cases, our design is not compatible with existing unmodified software. In particular, programs that rely on arithmetic or comparison of previously deallocated pointers may need to be changed slightly to deal with the fact that these pointers change. However, in cases where the difference is computed between two old pointers or where two old pointers are compared, no changes are needed because we merely set an unused bit. As such, we provide better compatibility than solutions such as DangNULL [37] that replace invalid pointers with a fixed value. An example where changes would be needed is the use of a hash table to rebase pointers after a call to `realloc`. It should be noted that the C standard explicitly states that pointer values become indeterminate when the object they point into reaches the end of its lifetime [3]. Nonetheless, we believe the strong protection offered by DangSan is sufficient to justify making these minor changes in the rare cases programs rely on undefined behavior.

DangSan is unable to track pointers that are copied in a type-unsafe way. Examples include pointers cast to integers and pointers copied by the `memcpy` function. The latter case is important whenever `realloc` is used to change the size of a buffer storing pointers. If the buffer cannot be resized in-place, `realloc` uses `memcpy` internally to move the contents of the buffer to their new location. As a consequence, one can no longer identify the copied values as pointers and invalidate them. This issue could be solved by hooking the `memcpy` function and copying over metadata for any pointers that are encountered. This can be done by looking up every pointer-sized value in a given chunk to determine whether it points to an object. However, we opted not to implement this strategy as it would greatly slow down `memcpy`, while we do not expect a large gain in detection coverage. Moreover, there would be a small risk of false positives unless we verify that the pointer locations are in the pointer log, increasing overhead even further. For these reasons, we believe it is not worthwhile to support type-unsafe copies of pointers. Similar to previous solutions [37, 51], our current DangSan prototype cannot automatically handle such copies.

We only track and invalidate pointers that are explicitly stored in memory. This means that there are two cases in which we do not track or invalidate pointers: pointers that are stored in registers and pointers that are spilled onto the stack by a function prologue to be restored to a register when the function returns to its caller. In practice, only the latter case is relevant because, when our free hook is called, the intermediate functions on the call stack have typically already spilled all preserved registers onto the stack. Unfortunately, however, there is no practical way to track such pointers because the callee does not know which registers were used by the caller to store pointers. As a consequence, these pointers are not tracked and will not be invalidated, resulting in the possibility of false negatives. To the best of our knowledge, this is a limitation shared with all pointer invalidation systems [37, 51], not just DangSan. A possible solution would be to walk the stack and check all values, but this comes with a risk of false positives. Moreover, it would be very inefficient if there is a deep call stack. We believe our design ultimately achieves a reasonable compromise.

In addition to the previous point, one can also consider the more complex case of an application with free-time race conditions, where one thread performs the free operation while another thread propagates the freed pointer via a register without proper synchronization. For example, consider the following sequence of events: thread A loads a pointer into a register, thread B then frees the object pointed to, and thread A then copies the pointer back into memory. In this case, the pointer will not be invalidated. The only solution to such race conditions in the application would be to stop all other threads whenever a free operation is performed and modify their state to invalidate pointers to the freed object in each thread's registers. This results in unacceptable overhead, especially since the worst-case impact of this race condition is just a false negative. As such, we consider this limitation inherent in pointer invalidation approaches. Nonetheless, our lockless approach makes the probability of this race condition slightly higher because the freeing thread may miss a newly propagated pointer added in a list slot that was already invalidated. In addition, our approach requires careful reuse of per-object metadata structures to prevent such race conditions from breaching metadata integrity. We believe our design is worthwhile as the risk of false negatives is inherent in pointer invalidation and our approach has the great benefit of preventing the need for synchronization in all cases.

8. Evaluation

In our evaluation, we compare programs instrumented by DangSan against a baseline configuration without instrumentation. We compiled both configurations with Clang 3.8 [36] using link-time optimization, which implies the maximum optimization level. We use the tcmalloc [24] allocator for both configurations to report unbiased results, given that tcmalloc generally introduces a speedup that should not be attributed to DangSan. We ran our benchmarks on Intel Xeon E5-2630 machines with 16 cores at 2.40 GHz and 128 GB of memory, running the 64-bit CentOS 7.2.1511 Linux distribution.

8.1 Effectiveness

To test whether DangSan can effectively mitigate use-after-free vulnerabilities in real-world applications, we instrumented three programs with known use-after-free vulnerabilities and ran exploits to determine whether they were still vulnerable. We specifically selected programs with publicly available exploits. In all cases, DangSan prevented the possibility of an attack.

For the OpenSSL [9] client we tested CVE-2010-2939 [5], a double free vulnerability with critical security impact. Rather than corrupting the client's memory, it caused the program to terminate prematurely:

```
src/tcmalloc.cc:290] Attempt to free invalid
pointer 0x80000000022ba510
```

The address in the message shows that our system correctly set the most significant bit of the dangling pointer and prevented the vulnerability from being exploitable.

We also tested CVE-2016-4077 [6] for Wireshark and a use-after-free vulnerability [8] on the Open Litespeed web server and found that, in both cases, dangling pointer invalidation crashes the program when the attacker tries to dereference the dangling pointer.

8.2 Performance and scalability

To measure the performance impact of DangSan, we ran a number of demanding CPU-intensive benchmarks, comparing the run time between an uninstrumented baseline version and a version protected by DangSan. In particular, we used the SPEC CPU2006 [31] benchmarking suite to test single-threaded performance and PARSEC [15] and SPLASH-2X [11] to test scalability to an increasing number of threads. We also tested the Apache 2.2.23 [1], Nginx 1.11.4 [7], and Cherokee [4] web servers with the ApacheBench [2] workload to show that our system works efficiently on modern multithreaded applications.

The performance results for the C and C++ programs of the SPEC CPU2006 benchmark suite are presented in Figure 9 and compared against the results reported by DangNULL [37] and FreeSentry [51]. The DangNULL results were estimated from the graph in the paper, while we received the raw data for an LLVM-based version of FreeSentry from its author. It should be noted that FreeSentry's numbers are measured with CIL, the platform the system is built on, as a baseline. Unfortunately, we were unable to rerun these systems on our own machine as no source is available for DangNULL and the public source for FreeSentry is less optimized than the one measured in the paper. However, the overheads should be reasonably comparable given that they are all normalized to a baseline run on the same system. DangNULL did not report results for libquantum, perlbench, dealII, and omnetpp. FreeSentry only reports a few SPEC CPU2006 benchmarks because they used the older CPU2000 where those were available. Unfortunately, however, these are not comparable to CPU2006 because CPU2000 uses simpler input files for its benchmarks. Our geometric mean over all benchmarks is 1.41, a slowdown of 41%. However, this includes omnetpp, a challenging memory-intensive benchmark that neither of the other systems evaluated. Therefore, to compare performance it is most appropriate to compare geometric means over intersection of the sets of benchmarks used by systems being compared. The geometric mean over the benchmarks run by DangNULL is a 55% slowdown while DangSan has a slowdown of just 22% over those same benchmarks. The geometric mean for FreeSentry is 30% and our own geometric mean over the same set of benchmarks is just 23%. Therefore, we conclude that we achieve much better performance than DangNULL, which also supports threads and does not track stack pointers, while we perform at par with FreeSentry despite its lack of support for multithreading.

We present scalability results for PARSEC and SPLASH-2X in Figure 10. Unfortunately, we could not run all benchmarks because many would not compile with LLVM due to nonstandard C/C++ language constructs. Note that this is not due to DangSan, as even the LLVM baseline does not compile. On almost all benchmarks, DangSan scales nearly as well as the baseline. The exceptions are Barnes and Canneal,

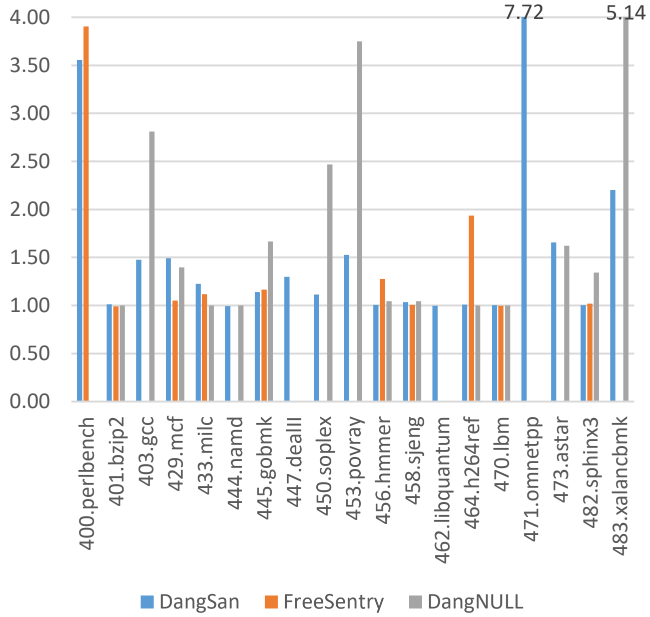

Figure 9. Performance overhead on SPEC CPU2006

where the baseline performance levels off while DangSan slows down as many threads are used. In practical situations, this would not make much of a difference as there is no reason to add threads if it does not improve the baseline performance (for example, with Barnes the baseline performs best at 32 threads with a runtime of 11.0s, but is already at 11.8s with 16 threads). There is a substantial negative overhead for Vips, which we believe may be due to systematic measurement error [38]. The geometric mean over the included benchmarks is 12.4% for one thread, and remains within the narrow range 17.3%-20.7% for 2-16 threads. It increases afterwards (29.7% for 32 and 33.6% for 64 threads), but it should be noted that the baseline is barely faster than for 16 threads in these cases on our system, which means that these configurations would be unlikely in practice. As such, we conclude that DangSan has reasonable overhead and good scalability on multithreaded workloads.

We benchmarked both web servers using Apachebench with the following settings: 128 concurrent connections in the client, 100000 requests issued, 32 worker threads and one worker process in the server. We have chosen this configuration to ensure a large amount of concurrency in the web server process to stress DangSan's multithreading ability. We transfer a very small file (44 bytes) locally to reduce I/O to a minimum and stress the CPU so as to provide a conservative estimate of the overhead incurred by DangSan. With Apache, we achieve 35634 requests per second on the baseline and 28274 when using DangSan. This corresponds with a slowdown of 21%. Nginx handles 11931 requests per second in the baseline and 8331 requests per second when using DangSan, corresponding with a slowdown of 30%. Cherokee handles 100155 requests per second in the baseline and 99841 with DangSan, a negligible slowdown. These numbers are similar to the results achieved on SPEC CPU2006 and show that DangSan is able to scale to programs with substantial concurrency.

8.3 Memory overhead

To determine the impact of our system on memory usage, we have measured the mean resident set size (RSS) while running the SPEC CPU2006 benchmark suite and the Apache and Nginx servers.

Figure 11 shows memory overhead for SPEC CPU2006. We have included memory overhead reported by DangNULL, while unfortunately FreeSentry does not report its memory overhead. The results show that pointer tracking can be very memory-intensive, especially for applications such as omnetpp that perform many allocations and keep many pointers to the allocated blocks of memory (see Table 1). This is the case for both DangSan and DangNULL. The geometric mean for DangSan is 2.4x memory overhead. The geometric mean for DangNULL is 2.3x, but they did not run some of the most memory-intensive benchmarks. The geometric mean for DangSan for the benchmarks reported by DangNULL is 1.8x. While this memory overhead is substantial, we improve overall memory overhead compared to the state of the art and we believe the overhead is worthwhile for the strong protection offered. Moreover, as we will show in Section 8.4, DangSan tracks and invalidates many more pointers than DangNULL does, so the memory overhead per pointer is actually much lower.

Figure 12 shows memory usage for PARSEC and SPLASH-2X measured as the maximum resident set size (RSS). Memory overhead differs greatly between benchmarks, but generally the number of threads does not have a strong impact. The main exception is water_nsquared, where memory overhead grows from 117.8% at one thread to 609.2% at 64 threads. This is due to the fact that it allocates a number of objects that is proportional to the number of threads and never frees most of them. While most benchmarks have low memory overhead, freqmine stands out for having 471.2% overhead, regardless of the number of threads. This is due to it propagating many pointers, even though the number of memory objects is not very large. As such, almost all of the memory is in the hash tables that hold these pointers. The geometric mean memory overhead over all benchmarks is 56.3% for a single thread, gradually grows to 66.6% for 16 threads and levels off. As such, we believe memory overhead not to be an issue for scalability to many threads.

To benchmark server memory overhead, we used the same configurations as in Section 8.2 and measured the mean RSS. Apache uses 40MB in the baseline configuration and 179MB when protected by DangSan. This means the memory overhead is 4.5x. For Nginx, the baseline uses 20MB while DangSan uses 36MB, resulting in an overhead of 1.8x. Cherokee uses 137MB in the baseline and 148MB

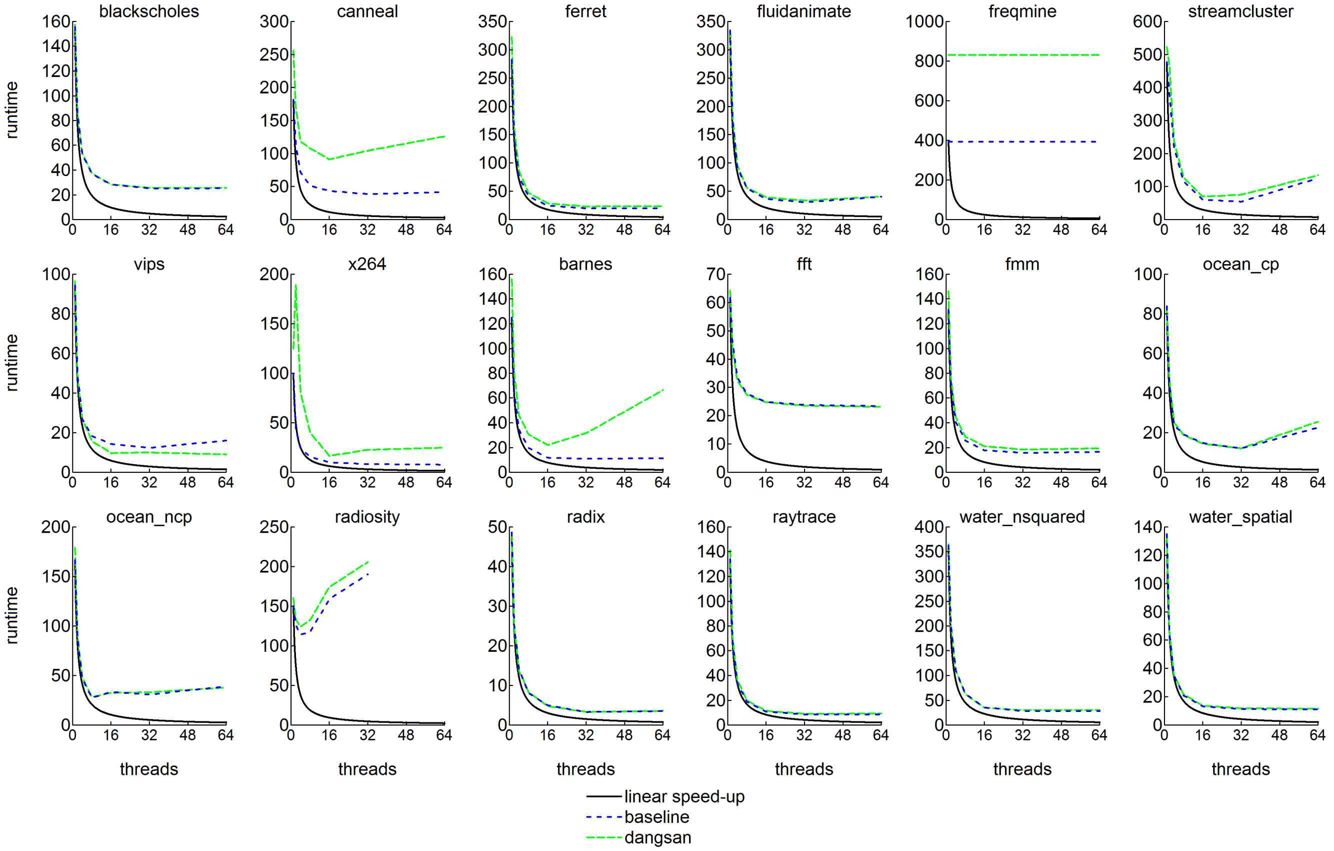

Figure 10. Scalability on PARSEC and SPLASH-2X

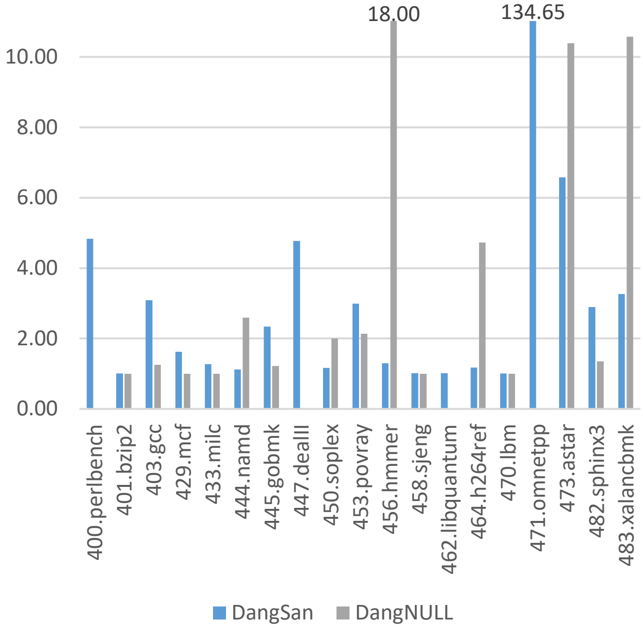

Figure 11. Memory overhead on SPEC CPU2006

with DangSan, corresponding with 1.1x. While the relative memory overhead on these servers is significant, it is important to note that in absolute numbers these amounts of memory should not matter much in typical server settings.

8.4 Coverage and statistics

Table 1 shows how many memory objects (# obj alloc) and pointers (# ptrs) we track and how many pointers we invalidated (# inval). We compare our results against DangNULL [37]. Unfortunately, no statistics are available for FreeSentry [51] to compare against. The table shows that, despite much lower overhead, we manage to invalidate many more pointers than DangNULL, providing better security. We manage to identify dangling pointers in nearly every program while DangNULL fails to do so in many cases, providing no additional security. Moreover, in all cases where both programs invalidate pointers, DangSan clears more than 100 times as many as DangNULL does. The differences are likely due to the fact that DangNULL can only track pointers stored on the heap while DangSan can track pointers stored in any area of memory. The fact that we offer both better performance and much better coverage demonstrates that our system can handle pointer registrations very efficiently.

In addition, Table 1 shows statistics about our approach itself. The number of hash tables allocated (# hashtable) is usually much smaller than the number of objects allocated. This means that many objects have only few pointers to them

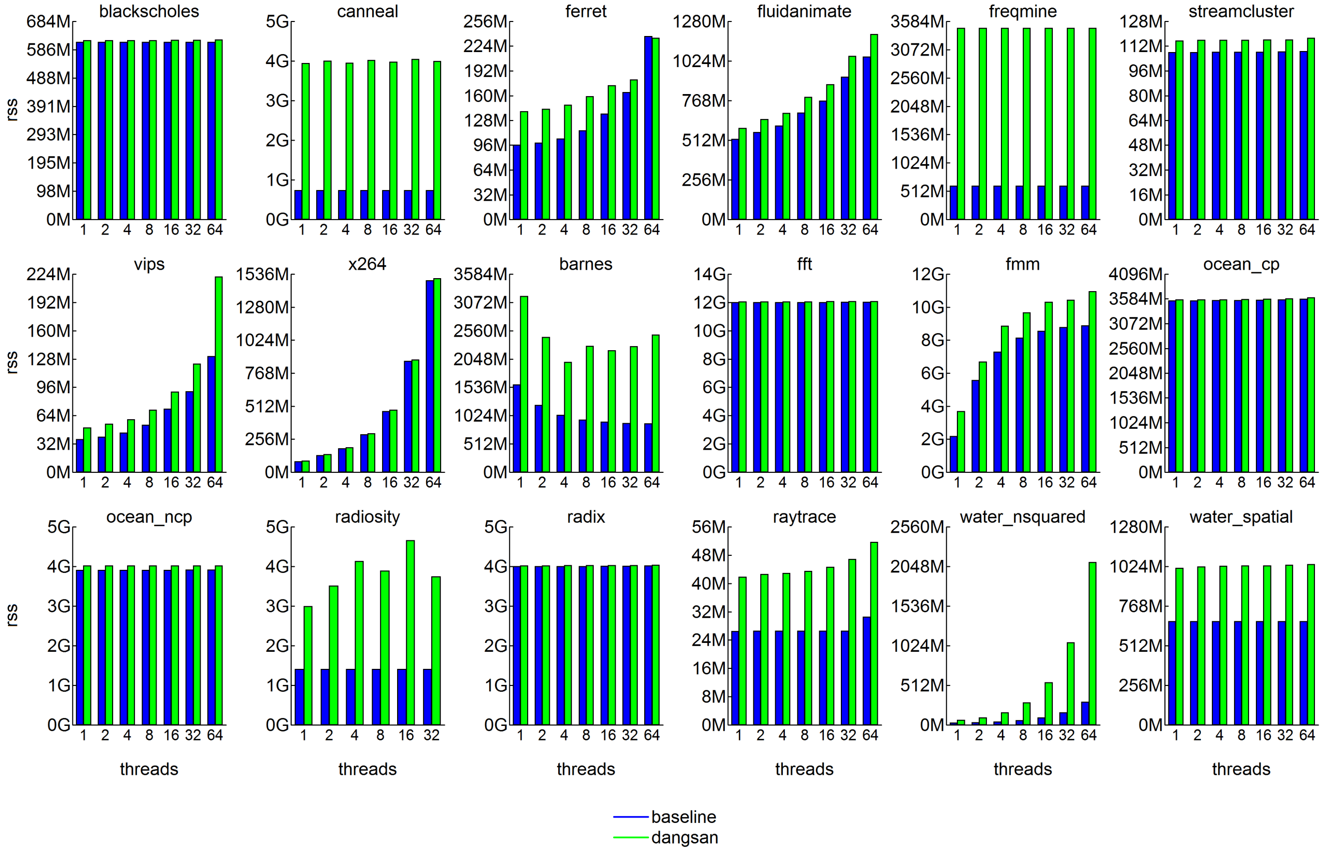

Figure 12. Memory usage on PARSEC and SPLASH2X

stored in memory, allowing them all to be stored in the static log allocated by default. The only benchmark that requires more than a million hash tables is omnetpp, which allocates a very large number of objects and stores a large number of pointers for almost half of those objects. This explains the relatively high memory overhead for this particular benchmark. The number of stale pointers listed in the table (# stale) indicates how many pointers were in the log for an object, but no longer referenced that object at free time. This differs greatly between benchmarks. milc stands out for having only six pointers invalidated, but almost a billion stale pointers. This means that pointers are often overwritten with pointers to other objects. In this case our design saves run-time overhead, because we do not need to remove the stale pointers from the log, but incurs additional memory overhead because we store pointers that are no longer relevant. Finally, the number of duplicate pointers (# dup) indicates how many pointers were prevented from being stored multiple times by our lookback and hash table. The extreme number of duplicate pointers for benchmarks such as perlbench shows that these mechanisms in our design are necessary to prevent near-unbounded memory consumption if only a log were used.

9. Related Work

There are several ways to automatically defend programs against use-after-free exploits: pointer invalidation systems, systems that check pointer dereferences, secure memory allocators, static analysis, taint tracking, and garbage collection. In this section we discuss each type of defense to compare their effectiveness and practicality against DangSan.

Pointer invalidation The closest to our design are DangNULL [37] and FreeSentry [51] which, like DangSan, keep track of pointers to each object and invalidate them once an object is freed. However, these systems have important limitations that hinder practicals adoption. DangNULL can only track pointers that are themselves embedded in heap objects, which means that it cannot invalidate pointers stored on the stack or in global memory. DangSan, on the other hand, is able to track pointers anywhere in memory. Another issue with DangNULL is the fact that it incurs considerably higher overhead compared to DangSan, despite the much smaller number of pointers it invalidates. Moreover, it uses data structures that require locking. Although the authors did not measure scalability, we expect DangSan's lock-free design to naturally scale much better to a large number of threads. FreeSentry [51] can track all pointers and offers run-time overhead comparable to ours, but it cannot sup-

benchmark	DangSan						DangNULL		
	# obj alloc	# hashtable	# ptrs	# inval	# stale	# dup	# obj alloc	# ptrs	# inval
400.perlbench	350m	380k	40490m	362m	53m	31557m			
401.bzip2	258	0	2200k	108	90	1868k	7	0	0
403.gcc	28m	524k	7170m	76m	110m	6738m	165k	3167k	14k
429.mcf	20	3	7658m	0	56m	7602m	2	0	0
433.milc	6530	6128	2585m	6	977m	1600m	38	0	0
444.namd	1339	0	2970k	3148	2159	1864k	964	0	0
445.gobmk	622k	15	607m	687k	46k	597m	12k	0	0
447.dealII	151m	49	117m	27m	3975k	4220k			
450.soplex	236k	18k	836m	2913k	45m	785m	1k	14k	140
453.povray	2427k	281	4679m	2218k	1565k	4457m	15k	7923k	6k
456.hmmer	2394k	56	3829k	1669k	100k	2040k	84k	0	0
458.sjeng	20	0	4	0	0	0	1	0	0
462.libquantum	164	0	130	16	49	30	49	0	0
464.h264ref	178k	271	11m	318k	125k	5164k	9k	906	101
470.lbm	19	0	6004	0	2	3002	2	0	0
471.omnetpp	267m	104m	13099m	36m	3421m	9207m			
473.astar	4800k	207k	1235m	11m	111m	1110m	130k	2k	20
482.sphinx3	14m	2910	302m	9880k	476k	280m	6k	814k	0
483.xalancbmk	135m	342k	2387m	152m	157m	1450m	28k	256k	10k

Table 1. Statistics for SPEC CPU2006.

port multithreaded programs, including important classes of vulnerable applications such as servers and browsers. While locking could be added to their design to preserve consistency of their shared data structures, it is to be expected that this would dramatically increase their overhead. DangSan, on the other hand, has been designed from the ground up to deal with concurrency efficiently.

Pointer dereference checking Some other systems to defend against dangling pointer exploits also track which regions are allocated, but perform checks when pointers are dereferenced rather than invalidating them when they are deallocated. CETS [39] attaches a label to each allocation and performs a check when the allocated pointer is dereferenced. While this approach is very complete, covering even cases where pointers are temporarily converted into integers, it imposes a higher run-time overhead compared to pointer invalidation systems. Moreover, it is prone to false positives and has poorer compatibility than competing solutions [37, 51]. For these reasons, we believe this class of detection systems to be less suitable for practical adoption.

Secure memory allocators Another class of systems that defends against various classes of memory errors including use-after-free are systems based on secure memory allocators. These systems do not track individual allocations but rather try to prevent allocated objects from ending up at the same address as previously freed objects. Typical examples are DieHard [14], DieHarder [40], Cling [12], and AddressSanitizer [45]. While these systems can effectively detect accidental use-after-free operations and some of them achieve reasonably low overheads, they are less suitable as detection systems against deliberate attacks. For example, Lee at al. [37] have shown that these systems allow the attacker to force the reuse of a freed memory region.

Static analysis While the approaches discussed so far operate at run time, it is also possible to use purely static techniques to detect use-after-free vulnerabilities. GUEB [23] is an example of this approach. The main benefit of using only static analysis is that no performance overhead is imposed during the execution. Unfortunately, however, purely static approaches can only recognize relatively simple cases and are therefore inherently prone to false negatives. Moreover, GUEB only targets small programs. While static approaches are very suitable as a way to find some vulnerabilities in a fully automated fashion, they need to be complemented by dynamic approaches to harden software against attacks based on residual vulnerabilities.

Taint tracking Approaches based on taint tracking do not use the programmer-assigned variable types to determine which values are pointers, but instead track the pointer value from its allocation site onward. An example of taint tracking being applied to dangling pointers is Undangle [19]. This approach has the benefit of avoiding the loss of metadata when pointers are copied in a type-unsafe way and can therefore achieve very good protection. Nevertheless, cases that cause taint propagation and thus dynamic pointer tracking to fail do exist in practice [46], resulting in incomplete use-after-free detection coverage. In addition, the taint tracking overhead is too high for production usage.

Garbage collection There are various safe programming languages that aim to replace the traditional C/C++ languages in such a way that common temporal memory errors are unlikely or even impossible. One typical example is Java [42]. To solve the problem of dangling pointers, these languages often rely on garbage collection rather than on explicit freeing of memory. As a consequence, dangling pointers cause the object they point to to remain allocated, converting the problem into a less exploitable memory leak instead. Rust [10] takes an intermediate approach, letting the programmer manage memory but providing smart pointer primitives that allow the programmer to do efficient garbage collection in specific cases. However, while these solutions are suitable for many newly created application programs, using them for existing software with precise garbage collection semantics requires a nontrivial porting effort for systems programs [43].

A more viable alternative is to use conservative garbage collection libraries tailored to C/C++ programs. For example, the Boehm garbage collector [16] provides a dedicated memory allocation API that offers conservative garbage collection. Although this alleviates the porting effort, semantic differences between the memory allocation interfaces may mean that it still requires a significant porting effort for large programs. In addition, compared to DangSan, conservative garbage collection is more prone to false positives due to type accuracy issues (although this risk is still small on 64-bit systems [32]) and can introduce other security problems, for example, providing attackers with a side channel (although others exist [18, 27, 41]) to bypass address space layout randomization (ASLR) [30]. Finally, garbage collection in general offers poor use-after-free detection guarantees, leaving a window of vulnerability between free and pointer invalidation operations.

10. Conclusion

Although use-after-frees are a major threat to systems security, no existing detection system is both practical, complete, and applicable to widespread multithreaded applications. To address all these concerns, we presented DangSan, a new use-after-free detection system based on pointer tracking and invalidation. Our design relies on efficient variable compression ratio memory shadowing and on scalable per-thread pointer logs inspired by the way log-structured file systems operate. As a result, DangSan is as efficient as existing detection systems that do not support multithreading, and much more efficient and scalable than state-of-the-art multithreaded solutions despite tracking and invalidating many more pointers. To foster further research in the field, we have made the source code of our DangSan prototype available as open source at `https://github.com/vusec/dangsan`.

Acknowledgments

We thank our shepherd, Pascal Felber, and the anonymous reviewers for their valuable feedback. This work was supported by the European Commission through project H2020 ICT-32-2014 “SHARCS” under Grant Agreement No. 64457 and the Netherlands Organisation for Scientific Research through grant NWO 639.023.309 VICI “Dowsing”.

References

[1] Apache. `https://www.apache.org/`, .

[2] Apachebench. `http://httpd.apache.org/docs/2.4/programs/ab.html`, .

[3] The ISO/IEC 9899:2011 (c11) standard. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853`.

[4] Cherokee web server. `http://cherokee-project.com/`.

[5] Cve-2010-2939: Double free vulnerability in the ssl3_get_key_exchange function in the openssl client. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2010-2939`, .

[6] Cve-2016-4077. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-4077`, .

[7] Nginx. `https://nginx.org/`.

[8] Open litespeed use after free vulnerabilit. `http://www.security-assessment.com/files/documents/advisory/Open%20Litespeed%20Use%20After%20Free%20Vulnerability.pdf`, .

[9] OpenSSL. `https://www.openssl.org/`, .

[10] Rust. `https://www.rust-lang.org/`.

[11] A memo on exploration of SPLASH-2 input sets. `http://parsec.cs.princeton.edu/doc/memo-splash2x-input.pdf`.

[12] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security*, 2010.

[13] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security*, 2009.

[14] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *PLDI*, 2006.

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[16] H. Boehm. A garbage collector for C and C++. `http://www.hboehm.info/gc/`.

[17] E. Bosman, A. Slowinska, and H. Bos. Minemu: The world’s fastest taint tracker. In *RAID*, 2011.

[18] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup Est Machina: Memory deduplication as an advanced exploitation vector. In *S&P*, 2016.

[19] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA*, 2012.

[20] X. Chen, H. Bos, and C. Giuffrida. CodeArmor: Virtualizing the code space to counter disclosure attacks. In *EuroS&P*, 2017.

[21] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *USENIX Security*, 2005.

[22] L. Deng, Q. Zeng, and Y. Liu. ISboxing: An instruction substitution based data sandboxing for x86 untrusted libraries. In *IISC*, 2015.

[23] J. Feist, L. Mounier, and M.-L. Potet. Statically detecting use after free on binary code. *JICV*, 10(3), 2014.

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

[25] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security*, 2012.

[26] E. Goktas, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos. Undermining information hiding (and what to do about it). In *USENIX Security*, 2016.

[27] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, 2017.

[28] I. Haller, E. van der Kouwe, C. Giuffrida, and H. Bos. METAlloc: Efficient and comprehensive metadata management for software security hardening. In *EuroSec*, 2016.

[29] I. Haller, J. Yuseok, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe. TypeSan: Practical type confusion detection. In *CCS*, 2016.

[30] A.-A. Hariri, B. Gorenc, and S. Zuckerbraun. Abusing silent mitigations: Understanding weaknesses within Internet Explorer's Isolated Heap and MemoryProtection. In *Black Hat USA*, 2015.

[31] J. L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4), 2006.

[32] M. Hirzel and A. Diwan. On the type accuracy of garbage collection. In *ISMM*, 2000.

[33] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer. Enforcing least privilege memory views for multithreaded applications. In *CCS*, 2016.

[34] Y. Jang, S. Lee, and T. Kim. Breaking kernel address space layout randomization with Intel TSX. In *CCS*, 2016.

[35] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos. No needto hide: Protecting safe regions on commodity hardware. In *EuroSys*, 2017.

[36] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[37] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[38] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS*, 2009.

[39] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: Compiler-enforced temporal safety for C. In *ISMM*, 2010.

[40] G. Novark and E. D. Berger. DieHarder: Securing the heap. In *CCS*, 2010.

[41] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida. Poking holes in information hiding. In *USENIX Security*, 2016.

[42] Oracle. Java. `https://www.java.com/`.

[43] J. Rafkind, A. Wick, J. Regehr, and M. Flatt. Precise garbage collection for C. ISMM, 2009.

[44] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1), 1992.

[45] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX ATC*, 2012.

[46] A. Slowinska and H. Bos. Pointless tainting? evaluating the practicality of pointer tainting. In *EuroSys*, 2009.

[47] A. Slowinska, T. Stancescu, and H. Bos. Body armor for binaries: Preventing buffer overflows without recompilation. In *USENIX ATC*, 2012.

[48] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[49] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *OSDI*, 2016.

[50] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in Linux kernel. In *CCS*, 2015.

[51] Y. Younan. FreeSentry: Protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.