A Robust and Efficient Defense against Use-after-Free Exploits via Concurrent Pointer Sweeping

Daiping Liu
University of Delaware
Newark, DE
dpliu@udel.edu

Mingwei Zhang
Intel Labs
Hillsboro, OR
mingwei.zhang@intel.com

Haining Wang
University of Delaware
Newark, DE
hnw@udel.edu

ABSTRACT

Applications in C/C++ are notoriously prone to memory corruptions. With significant research efforts devoted to this area of study, the security threats posed by previously popular vulnerabilities, such as stack and heap overflows, are not as serious as before. Instead, we have seen the meteoric rise of attacks exploiting use-after-free (UaF) vulnerabilities in recent years, which root in pointers pointing to freed memory (i.e., dangling pointers). Although various approaches have been proposed to harden software against UaF, none of them can achieve robustness and efficiency at the same time. In this paper, we present a novel defense called pSweeper to robustly protect against UaF exploits with low overhead, and pinpoint the root-causes of UaF vulnerabilities with one safe crash. The success of pSweeper lies in its two unique and innovative design ideas, concurrent pointer sweeping (CPW) and object origin tracking (OOT). CPW exploits the increasingly available multicores on modern PCs and outsources the heavyweight security checks and enforcement to dedicated threads that can run on spare cores. Specifically, CPW iteratively sweeps all live pointers in a concurrent thread to find dangling pointers. This design is quite different from previous work that requires to track every pointer propagation to maintain accurate point-to relationship between pointers and objects. OOT can help to pinpoint the root-causes of UaF by informing developers of how a dangling pointer is created. We implement a prototype of pSweeper and validate its efficacy in real scenarios. Our experimental results show that pSweeper is effective in defeating real-world UaF exploits and efficient when deployed in production runs.

ACM Reference Format:
Daiping Liu, Mingwei Zhang, and Haining Wang. 2018. A Robust and Efficient Defense against Use-after-Free Exploits via Concurrent Pointer Sweeping . In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3243734.3243826

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243826

1 INTRODUCTION

Memory corruption vulnerabilities have plagued software written in low-level languages like C/C++ for decades. On one hand, effective defenses against previously popular attacks, such as stack and heap overflows [16, 20, 22, 23, 25, 33, 39, 42, 47, 50, 59], have been developed and deployed in commodity systems, making the exploitation of such memory corruption bugs much harder in system software (e.g., browser or operating system). On the other hand, recent years have seen the meteoric rise of memory corruption attacks exploiting use-after-free (UaF) vulnerabilities that root in pointers pointing to deallocated memory (i.e., dangling pointers). Actually, UaF vulnerability has become the largest and severest on-going exploit vector in numerous popular applications [36].

Different approaches have been proposed to harden the memory safety of software against UaF vulnerabilities. Most of the existing solutions attempt to disrupt UaF exploits by making an explicit [40, 43, 45, 51, 56] or implicit [26] safety check on every pointer dereference. An alternative approach is to reshape memory allocators to avoid unsafe memory reuse [14, 16, 47]. Conservative garbage collection [6, 18] heads off UaF exploits through automatic memory management. Moreover, the Silicon Secured Memory (SSM), recently shipped in Sparc M7 processors, implements tagged memory as a hardware UaF defense [5]. Recent works [19, 36, 58] track pointer propagation and nullify dangling pointers at object free.

Unfortunately, these solutions still suffer two main drawbacks. First, robustness and efficiency cannot be achieved at the same time. UaF exploits are guaranteed to be defeated but usually with unacceptable or unpredictable overhead [6, 26, 36, 43, 56, 58]. Systems like Cling and SSM incur trivial overhead but provide only partial [14] or probabilistic [5, 16, 47] memory safety. Second, software developers usually cannot obtain sufficient information about the exploited UaF vulnerabilities during production runs, making it difficult to debug and craft patches.

This paper presents **pSweeper**, a novel defense system that effectively protects against UaF exploits and imposes low overhead for deployment in production environments, as well as pinpoints the root-causes of UaF vulnerabilities for easier and faster fixing. The basic protection principle of pSweeper is to proactively neutralize dangling pointers so as to disrupt potential UaF exploits, which is similar to DANGNULL, DangSan, and FreeSentry [36, 54, 58]. However, very different from those previous solutions, pSweeper has two unique and innovative features, concurrent pointer sweeping and object origin tracking, to overcome the above shortcomings.

In order to find and neutralize dangling pointers, existing approaches [36, 54, 58] *synchronously* keep track of pointer propagation at runtime to maintain accurate point-to relationships between

pointers and objects. This design can incur undue overhead, e.g., 80% in DANGNULL [36]. Leveraging the increasingly available multi-cores on PCs, pSweeper instead explores a very different design, **concurrent pointer sweeping (CPW)**, which *iteratively* sweeps all live pointers in *concurrent* threads and neutralizes dangling pointers. Compared to existing work, a major difference and advantage of this design is that, we only need to check if a pointer is dangling (i.e., pointing to freed memory) when pSweeper threads sweep it. In particular, we do not need to know which object a pointer is pointing to. Thus, there is no need for us to maintain accurate point-to relationship anymore, which accounts for the most overhead in previous approaches. In general, using a spare CPU core, pSweeper can effectively reduce the latency induced by security checks that are instrumented to applications.

The main challenge of implementing CPW lies in identifying and efficiently handling entangled races among pSweeper and application threads. For example, since pSweeper has to check live pointers one by one, a dangling pointer may propagate to an already-checked pointer before being neutralized. pSweeper would provide incomplete protection if such races were left unhandled. While most race conditions can be addressed by using heavyweight synchronization mechanisms like locks, it could cause unacceptable overhead and thus offset the design benefits of CPW. To this end, we devise several simple and efficient mechanisms to correctly handle these race conditions, which represent our major technical contributions. Our key design principle is to place heavyweight workload on pSweeper threads and instrument as less code as possible to application threads. In particular, we leverage hardware features and lock-free algorithms to avoid stalling application threads whenever possible.

Another desirable feature that pointer neutralization can provide is object origin tracking (OOT). When software crashes due to dangling pointer dereference, OOT can inform us of how the dangling pointer is caused, i.e., where the pointed object is allocated and freed. This information can greatly help programmers pinpoint the root-causes of UaF vulnerabilities. pSweeper achieves OOT by encoding origin information into neutralized dangling pointers. Like ASAN [51], pSweeper can pinpoint root-causes of UaF vulnerabilities in one safe crash. However, pSweeper achieves this at a trivial cost.

Finally, we implement a prototype of pSweeper and demonstrate its effectiveness using real-world UaF vulnerabilities. Our evaluation results on SPEC CPU2006 benchmarks show that the induced overhead is quite low (12.5%~17.2% compared to around 40% by state-of-the-art). We demonstrate that pSweeper scales quite well on multi-thread applications using PARSEC benchmarks. We further conduct two case studies with Lighttpd web server and Firefox browser.

The remainder of this paper is organized as follows. Section 2 introduces the background of UaF. Section 3 describes the overview of pSweeper. Section 4 presents the detailed design of pSweeper. Section 5 empirically evaluates the effectiveness and performance of pSweeper. Section 6 discusses the limitations of pSweeper. Section 7 surveys related work, and finally, Section 8 concludes the paper.

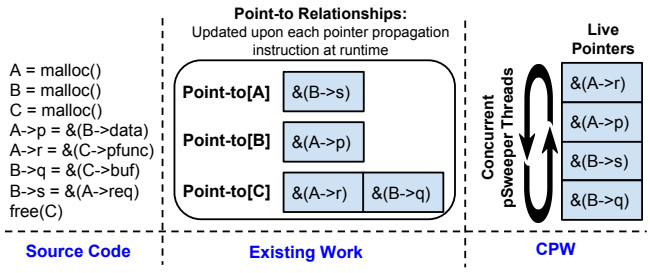

Figure 1: Comparison of CPW with existing solutions [36, 54, 58]. *Previous approaches require to maintain complete point-to relationships between pointers and objects. Whenever a pointer points to a different object, the point-to relationship must be updated. Instead, CPW in pSweeper only records live pointers and memory allocation status. In particular, CPW does not need to know which object a pointer is pointing to or how many different objects a pointer has pointed to. CPW only needs to check if a pointer is dangling.*

2 BACKGROUND AND THREAT MODEL

Dangling Pointer. A pointer variable `p` is dangling *iff* an object `O` with address range ∀ m, size : `[m, m+size-1]` has been freed and `p∈[m, m+size-1]`.

In practice, UaF exploits commonly reuse freed memory and fill it with specially crafted contents which are then accessed through dangling pointers. Therefore, it is insufficient to check whether a pointer points to freed memory. Instead, it is imperative to enforce that dangling pointers never point to memory that can be arbitrarily manipulated by attackers.

Threat Model. This paper focuses on UaF vulnerabilities rooted in dangling pointers which can point to any memory region including heap, stack, code and data. The attacker can crash applications but cannot cause any other consequences. Spatial attacks that exploit out-of-bound writes like buffer overflows, and temporal attacks that exploit uninitialized reads, are out of our scope. Therefore, similar to related works [36, 54, 58], we do not protect our system from these vulnerabilities and it should be used along with orthogonal protectors. We also assume applications do not have concurrency bugs. Finally, we do not deal with undefined behaviors, such as `delete` objects created using `new[]`.

3 OVERVIEW

3.1 High-Level Approach of pSweeper

pSweeper aims to robustly protect against UaF exploits with low overhead and pinpoint the root-causes of UaF vulnerabilities being exploited in the wild. To accomplish these, pSweeper proposes Concurrent Pointer Sweeping (CPW) and Object Origin Tracking (OOT). pSweeper follows a similar protection principle to pointer nullification in DANGNULL [36], FreeSentry [58] and DangSan [54]. In particular, when an object is freed, all dangling pointers are neutralized to disrupt UaF exploits. However, pSweeper differs significantly in two key design aspects:

1. How to find dangling pointers; and
2. What value is used to neutralize dangling pointers.

Finding Dangling Pointers. Previous approaches [36, 54, 58] *synchronously* maintain accurate point-to relationships between pointers and objects at runtime, so that when an object is freed, they can locate the set of pointers that are still pointing to the freed object and nullify them. In such a design, for each pointer propagation instruction, they need to track (1) in which object this pointer is located and (2) which object this pointer is pointing to. Since a pointer can point to the middle of an object, both operations require range-based searches that are notoriously expensive. Even worse, synchronization mechanisms like locks should be used in several places to avoid races among application threads [36].

We propose CPW, a totally different design. The key feature of CPW is to decouple the search for dangling pointers from application code. Specifically, CPW *iteratively* sweeps all live pointers at runtime in *concurrent* threads to neutralize the dangling pointers. Since pSweeper scans every pointer at runtime to find dangling pointers, there is no need to maintain complete point-to relationships anymore. Figure 1 illustrates the differences between CPW and previous solutions [36, 54, 58]. Note that, in principle, pSweeper does not improve the performance of individual security checks. Instead, pSweeper leverages spare CPU cores to reduce the latency of security checks that are instrumented into applications, and it may consume more CPU resources than DANGNULL, FreeSentry, and DangSan.

Since pSweeper works in a concurrent manner, a freed memory block may get re-allocated before pSweeper has eliminated current dangling pointers. This may cause two problems. First, UaF vulnerability exists in the time window between memory free and pSweeper's dangling point neutralization, which could be exploited by attackers to hijack control flow or escalate privileges after gaining the control of the re-allocated memory block. Second, once a memory block is reused, it becomes impossible for pSweeper to precisely find all dangling pointers. To prevent the occurrence of UaF vulnerable windows and avoid missing dangling pointers, CPW defers object frees to the end of every round of sweeping. Therefore, for each object, CPW introduces a grace period $T_{grace} = [T_{issue}, T_{release}]$, where T_{issue} is when application code issues a free request and $T_{release}$ is the actual time when CPW releases the memory back to OS.

Choosing Value for Pointer Neutralization. Previous works [36, 54, 58] simply set dangling pointers to `NULL` or kernel space. This guarantees that applications crash safely when dangling pointers are accessed.

pSweeper instead specially crafts the values to neutralize dangling pointers. Our key insight is that the crucial information to pinpoint root-causes of UaF vulnerabilities is how a dangling pointer is caused, i.e., how the pointed object is allocated and freed. Therefore, besides enforcing safe crash upon dangling pointer dereference, pSweeper also encodes object origin information into dangling pointers to achieve OOT. Compared with other tools that provide a similar feature to OOT [48, 51], pSweeper is more efficient.

Enforced Protection Protocol. Building upon CPW and OOT, pSweeper will enforce the runtime protection protocol as follows. Given a dangling pointer `p`:

- If `p` is accessed before being neutralized, applications continue to execute *correctly* similar to conservative garbage collection [31], because memory free requests are deferred by pSweeper.
- If `p` is accessed after being neutralized, applications abort *safely* with object origin information dumped.

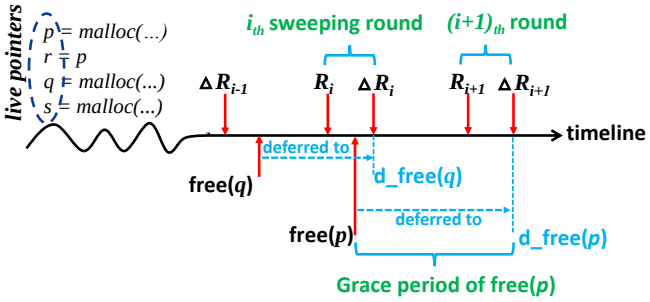

Figure 2: Illustration of pSweeper in time line.

3.2 An Illustration Example

Figure 2 illustrates pSweeper with an example in time line. All `malloc()`, `free()`, and assignment instructions are executed in application threads. R_i and ΔR_i denote the start and end of the i_{th} sweeping round of pSweeper threads, respectively.

Assume the application executes three `malloc()` and one pointer assignment before ΔR_{i-1}. From these instructions, pSweeper identifies four live pointers, `p`, `r`, `q`, and `s` at runtime.

During the interval of sweeping rounds, i.e., between ΔR_{i-1} and R_i, an application thread invokes `free(q)`. However, this free request will be hooked by pSweeper and delayed to the end of i_{th} sweeping round. During the i_{th} sweeping round, pSweeper checks all four pointers to find and neutralize the dangling one q. At ΔR_i, the delayed `free(q)` gets executed. If another memory block p is freed during the i_{th} round, it will be delayed to ΔR_{i+1}.

While the overall approach sounds simple, it is non-trivial to efficiently handle the entangled races among pSweeper and application threads. For instance, during the i_{th} sweeping round, assume pSweeper has checked p and r but has not neutralized q. It is possible that an application thread propagates the dangling pointer to a swept one, e.g., executing $r = q$. pSweeper must efficiently handle such cases.

3.3 Architecture of pSweeper

To implement CPW and OOT, pSweeper combines compile-time instrumentation and a runtime library, as shown in Figure 3. There are three components in pSweeper:

Pointer address identification. pSweeper first statically identifies pointer variables so that pointer addresses can be located at runtime (§4.2). It achieves this by analyzing the types of local/global variables. For pointers in dynamically allocated objects (on heap), we adopt the same strategy as previous work [36, 54, 58]. Specifically, we rely on the types of operands in `store` instructions. For each pointer `store` instruction, a snippet of code is instrumented into applications, which will bookmark live pointers at runtime.

Concurrent pointer sweeping thread. At runtime, dedicated pSweeper threads iteratively sweep all live pointers and neutralize

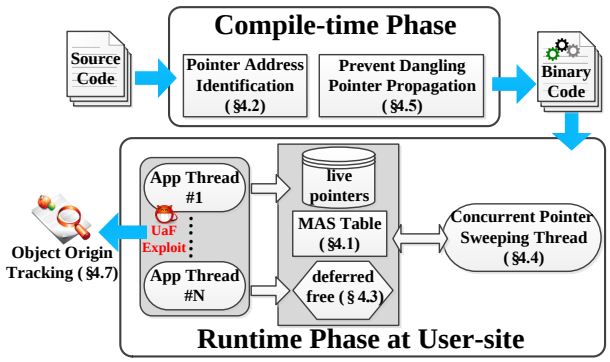

Figure 3: The architecture of pSweeper.

API	Parameter	Description
setFreed(*addr*)	A virtual address	Set the freed flag for *addr*.
clearFreed(*addr*)	A virtual address	Clear the freed flag for *addr*.
isFreed(*addr*)	A virtual address	Return TRUE if *addr* is not allocated.
setPointer(*addr*)	A virtual address	Set pointer flag for *addr*.
clearPointer(*addr*)	A virtual address	Clear pointer flag for *addr*.
isPointer(*addr*)	A virtual address	Return TRUE if *addr* is a pointer.

Table 1: Auxiliary APIs used in pSweeper.

the dangling ones. The asynchronous nature of CPW requires object frees to be deferred (§4.3). Otherwise, when a memory block is freed, it may get reused before CPW threads can neutralize all dangling pointers. The main challenge of CPW lies in efficiently handling races among pSweeper and application threads. In particular, dangling pointers can propagate during concurrent sweeping. To this end, we devise a simple and efficient mechanism to prevent dangling pointer propagation (§4.5) so that dangling pointers are guaranteed to be neutralized in one single round of sweeping.

Object origin tracking (OOT). Finally, pSweeper encodes object origin information into dangling pointers so that once they are dereferenced, pSweeper can inform developers how corresponding objects are allocated and freed (§4.7).

4 SYSTEM DESIGN AND IMPLEMENTATION

In this section, we detail the design of pSweeper. Due to the asynchronous design, we need to efficiently handle the entangled races between application and pSweeper threads. In particular, we aim to address these races with lock-free algorithms, which can highly correlate with the memory model of multicore processors. Our current design is built upon the memory model of x86 [7], AMD64, and SPARC[1].

4.1 Memory Allocation Status Table

To clearly describe the design of pSweeper's core components, we first define a list of APIs in Table 1. The first set of three APIs track the allocation status of a virtual address. The other set of three APIs can facilitate pSweeper to efficiently determine if a virtual address contains a pointer.

[1]The default mode of SPARC is Total Store Order (TSO).

Since several components of pSweeper rely on the APIs in Table 1, the efficiency of these APIs is critical to the performance of pSweeper. We now describe how these APIs are implemented. A straightforward way is to check against the list of live objects and pointers. However, for `isFreed()`, this simple implementation runs in time complexity of $O(P * M)$ for `isFreed()` and $O(P)$ for `isPointer()`, where P and M denote the number of live pointers and the number of freed objects, respectively. Obviously, this naive implementation does not scale well.

To this end, we design a memory allocation status (MAS) table, which is a shadow heap similar to the design philosophy in previous work [34, 39, 42, 54, 55]. The MAS table is built on the fact that pSweeper only needs to know if a memory address is allocated or freed, and it does not need to know where the object boundaries are. Every byte in the MAS table records whether the corresponding byte on heap is allocated. As a result, pSweeper can achieve a fast check with one single memory read.

However, this implementation is still inefficient. First, it incurs high overhead to set and clear shadow heap in `setFreed()` and `clearFreed()`. Second, it doubles memory consumption. To optimize, we leverage the observation that pragmatic memory allocators usually enforce object size and alignment. For example, the base and size of small and large objects (based on a predefined size threshold) are usually aligned to multiples of the pointer and page size, respectively. Therefore, the MAS table only requires 1-byte for every page or 8-byte on x64 (4-byte on x86).

For `isPointer()`, `setPointer()`, and `clearPointer()`, we implement a pointer location mark (PLM) table similar to the MAS table. PLM represents every 8-byte on x64 and 4-byte on x86 to 1-byte, but PLM cannot compress a memory page to 1-byte.

4.2 Locating Live Pointers

pSweeper first statically identifies pointer variables at compile time, and instrument code to bookmark live pointers at runtime. Pointers can be on stack, data, and heap segments. Dangling pointers on all three regions can be exploited.

4.2.1 *Pointers on Data Segment.*

Pointers can reside in data segments, including global and static variables[2]. These pointers can generally be identified at compile time. For each global pointer variable, we instrument a store instruction to log its address to a buffer denoted as `globalptr`. `globalptr` is library-specific, i.e., every library as well as the executable has a dedicated buffer. pSweeper instruments `.init` and `.fini` sections to every executable and library so that `globalptr` is (de)allocated upon (un)loading.

4.2.2 *Pointers on Stack.*

pSweeper handles the pointers in function parameters and local variables in a similar way as global variables. However, due to the asynchronous design of pSweeper, pointers on stack need to be specially handled. Consider the dangling pointer p in Figure 4, before pSweeper neutralizes p, function $func1$ returns and $func2$ is subsequently invoked. Previously storing pointer p, the stack slot now contains a non-pointer variable i. If i by chance has a value

[2]We use "global variables" for short in the remainder of the paper.

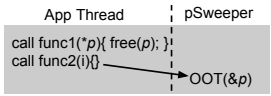

Figure 4: Race conditions of pointers on stack.

Algorithm 1 Compile-time store instruction instrumentation for pointer address identification.

```
1: function BOOKMARK_HEAP_POINTER( )
2:     for each storeinst do
3:         if onDataOrStack(storeinst.dest) then
4:             continue
5:         Instrument Bookmark_Ptr() after storeinst.
```

equal to the address of a freed memory slot, pSweeper can falsely neutralize it and thus corrupt application data.

To efficiently handle this race, pSweeper relocates all pointers on stack to a dedicated stack denoted as `stackptr`. In this way, every variable in `stackptr` is of pointer type. Thus, pSweeper can safely sweep and neutralize them. However, pointers in complex data types like `struct` and `class` cannot be easily moved to `stackptr` without losing compatibility. We therefore simply allocate all such variables on the heap.

4.2.3 *Pointers on Heap.*

Similar to previous works [36, 54, 58], we also rely on the types of operands in `store` instructions to track pointer addresses at runtime. The main difference lies in what task is performed at each pointer `store` instruction. All previous systems require to synchronously track (1) in which object a pointer is located and (2) which object a pointer is pointing to. This inevitably incurs high overhead due to the expensive range-based searches. In contrast, pSweeper simply bookmarks the addresses of live pointers. This, however, is still non-trivial to implement efficiently.

An assignment operation `LHS=RHS` is usually transformed to a compiler intermediate representation (IR) `store <ty> <val>, <ty>* <ptr>`, where `val` is the value in `RHS` and `ptr` is the memory address of `LHS`. If the type `<ty>` of `val` is a pointer, `LHS` is a pointer. However, its address should not be naively bookmarked for three considerations. First, we must ensure the pointer is not on data or stack segments. Second, we should ignore duplicate bookmarks for the same pointer. Finally, the pointer might be in a freed object. Algorithm 1 and 2 show how pSweeper bookmarks live pointers.

Excluding global/local pointers. We exclude global/local pointers in two steps. First, we identify `store` instructions for non-heap pointers at compile time and do not instrument them (Algorithm 1). Second, at runtime, we check if the address of a pointer is indeed in range of heap (Line 2 Algorithm 2).

Skipping duplicate bookmarks. Once a new pointer is encountered, we set the pointer flag using the API `setPointer(&ptr)`. In this way, when the pointer is encountered again, we can simply omit it.

Validity of pointer address. We next use the API `isFreed()` to check if the object where the pointer is contained has been freed (Line 7 Algorithm 2). Note that, there is a potential race that the object where `ptr` is contained gets freed and reused by another application thread after the check but before Line 10 Algorithm 2. We discuss this race further in §4.4.

Algorithm 2 Bookmark live pointer addresses.

PtrList: `live pointer list`

```
1: function BOOKMARK_PTR( &ptr )
2:     if notOnHeap(&ptr) then
3:         return
4:     if isPointer(&ptr) then
5:         return
6:     setPointer(&ptr)
7:     if isFreed(&ptr) then
8:         clearPointer(&ptr)
9:         return
10:    appendToList(&ptr, PtrList)
```

```
1 struct LiveObjNode{
2   obj_addr;   // object address
3   freeflag;   // Section §4.3
4   scanflag;   // Section §4.4
5   slotid;     // Section §4.7
6   struct LiveObjNode *prev, *next;
7 };
```

Figure 5: Metadata of live objects.

Live pointer list. We simply use a double-linked list (`PtrList`) to maintain all live pointers. As a result, `appendToList()` is quite efficient. Further, in order to avoid races among application threads which can concurrently operate on `PtrList`, we use a separate list for each thread. Note that, since we assume no concurrency bugs in applications, it's impossible that different application threads concurrently invoke appendToList() for the same pointer. Therefore, a thread-local `PtrList` is safe.

Removing stale pointers. Here we have described how to bookmark live pointers. When an object is freed, all pointers contained in it should be removed from `PtrList`. This is achieved in CPW (§4.4).

4.3 Deferred Free

pSweeper requires object frees to be deferred to the end of a sweeping round. To this end, pSweeper maintains live objects in a double-linked list (`ObjList`) and adds metadata `freeflag` for each object (Figure 5). In the hooked `malloc()`, pSweeper first sets `freeflag` to zero (Line 5 Algorithm 3) and then appends the new object to `ObjList` (Line 6 Algorithm 3). When `free()` is invoked in applications, we simply set `freeflag` as in Algorithm 4. Similar to `PtrList`, each application thread uses a thread-local list to maintain objects and nodes in `ObjList` are removed by CPW (§4.4).

4.4 Concurrent Pointer Sweeping (CPW)

CPW consists of two components, dedicated CPW threads and dangling pointer propagation instrumentation. Dedicated CPW threads are the core of CPW and they iteratively sweep live pointers to find and neutralize dangling ones. One challenge here is that application threads can propagate dangling pointers to the pointers that have been neutralized by CPW threads. We devise a simple and efficient mechanism (§4.5) to prevent dangling pointer propagation

Algorithm 3 Hooked malloc().

```
1: function MALLOC( size )
2:     obj ← real_malloc(size)
3:     clearFreed(obj)
4:     obj.scanflag ← 0
5:     obj.freeflag ← 0
6:     appendToObjList(obj, ObjList)
7:     mfence                          ▷ Memory barrier
```

Algorithm 4 Deferred free() invoked in applications.

```
1: function FREE( obj )
2:     assertDoubleFree(obj)           ▷ Abort upon double free.
3:     setFreed(obj)
4:     obj.freeflag ← 1
```

in application threads. Next, we describe each component in details. We first assume one CPW thread is spawned for a multi-threaded application and extend to multiple CPW threads in §4.6.

Algorithm 5 presents the pseudocode of CPW thread whose body is an infinite loop (Line 2) implementing iterative sweeping. CPW takes a list of live objects and pointers as input. In every round of sweeping, CPW threads execute in three steps.

- **Step 1** (Lines 4 ~ 9)

This step traverses live object list and if an object's `freeflag` is set, another field of metadata `scanflag` is set. `scanflag` is initialized as 0 in `malloc()` (Line 4 Algorithm 3). `fillWithSlotIndex()` is used by OOT which will be described in §4.7. Step 1 is required to guarantee that an object whose `freeflag` is set during pointer sweeping is not prematurely freed.

- **Step 2** (Lines 11 ~ 18)

This step sweeps all live pointers and checks if a pointer is dangling (Line 15). Dangling pointers are then neutralized with a value containing object origin information (Line 16). However, this step has a time of check to time of neutralization race as illustrated in Figure 6. To be specific, the value of `p` can be modified by application threads after the `isDangling()` check.

To address this, we observe that if a dangling pointer is modified by application threads between `isDangling()` and `OOT()`, we should preserve the value written by application threads and the neutralization by pSweeper can fail safely. On the one hand, if the new value written by application threads points to a live object, the dangling pointer is eliminated by application threads and we must preserve the value for correct execution. On the other hand, if the new value points to a freed object, this propagation will be handled by our mechanism that prevents dangling pointer propagation (§4.5). Fortunately, modern processors provide efficient hardware instructions such as `lock cmpxchg` that exactly meet our needs.

In addition, CPW threads skip stale pointers, i.e., whose containing objects have been freed, and remove them from `PtrList` (Lines 12~14). To demonstrate that the race mentioned in §4.2.3 does not cause failures in CPW, we consider two cases.

Case 1: Line 7 in Algorithm 2 returns true. In this case, pSweeper always correctly skips stale pointers. In particular, no live pointer is

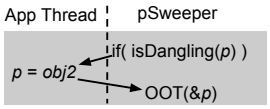

Figure 6: Time of check to time of neutralization race.

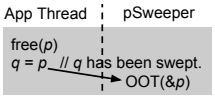

Figure 7: Dangling pointers can propagate to swept ones in application threads.

missed when `objFreed()` returns true but the memory has been allocated in a different application thread (due to inconsistency MAS table seen by different cores). This is because the `store` instruction must be executed after the hooked `malloc()` has returned. Otherwise, there is a concurrency bug in applications, which violates our assumptions in §2. Line 7 Algorithm 3 enforces that `objFreed()` must return false when the hooked `malloc()` returns.

Case 2: Line 7 in Algorithm 2 returns false. The only problem here lies in the possibility that the object where `ptr` is contained can get freed and reused before Line 10 Algorithm 2. In this case, the stale pointer will be appended to `PtrList`. However, this can happen only if there is a concurrency bug in applications, which violates our assumptions in §2.

- **Step 3** (Lines 19 ~ 25)

CPW threads now traverse object list again to free objects whose `scanflag` is set and remove them from the list. In order to avoid locks between insertion by applications and deletion by CPW, the tail node in `ObjList` is delayed until more nodes have been appended.

Avoiding endless sweeping rounds. Since new objects and pointers are created continuously by application threads, the `while`-loops in the above three steps may not terminate if they are not handled specially. To this end, CPW threads enforce that every round of sweeping terminates at the tail nodes of the lists (Lines 8, 17, and 25) that are recorded at the beginning of the loops (Lines 3 and 10). For Step 1 and 3, this enforcement is required because only objects that have been checked against every live pointer can be safely freed. For Step 2, this strategy is correct and safe because pSweeper prevents dangling pointer propagation (§4.5) and thus newly added pointers can be deemed as already swept.

4.5 Preventing Dangling Pointer Propagation

As shown in Algorithm 5, CPW threads sweep every live pointer only once in each round. Unfortunately, dangling pointers can propagate to the swept ones in application threads, as illustrated in Figure 7. Basically, pointers can propagate in three ways, assignment (e.g., `q = p`), function arguments (e.g., `func(p)`), and returns (e.g., `p = getPtr()`). In this section, we only describe how pointer assignment is handled because the other two ways are essentially also assignments.

Figure 8 presents how pSweeper handles pointer assignments. A pointer assignment `q = p` is usually compiled to two instructions, one `load` of `p`'s value and one `store` to `q`. To handle this case, we

Algorithm 5 Concurrent Pointer Sweeping (CPW) threads.

ObjList: `live object list`
PtrList: `live pointer list`

```
1:  function isDangling(p)
2:      if p is neutralized then
3:          return FALSE
4:      if isFreed(p) then
5:          return TRUE
6:      else
7:          return FALSE
8:  function CPW_Thread( )
9:      while True do
10:         objEnd ← getObjectListTail(ObjList)
11:         while obj ← getNextObj(ObjList) do
12:             if obj.freeflag then
13:                 obj.scanflag ← 1
14:                 fillWithSlotIndex(obj, obj.slotid)
15:             if obj == objEnd then
16:                 break
17:         ptrEnd ← getPtrListTail(PtrList)
18:         while ptr ← getNextPtr(PtrList) do
19:             if objFreed(&ptr) then
20:                 removePtr(&ptr, PtrList)
21:                 continue
22:             if isDangling(ptr) then
23:                 OOT(&ptr)
24:             if ptr == ptrEnd then
25:                 break
26:         while obj ← getNextObj(ObjList) do
27:             if obj.scanflag then
28:                 real_free(obj)
29:                 removeObj(ObjList, obj)
30:                 clearPLMTable(obj)
31:             if obj == objEnd then
32:                 break
33:         Sleep(t)                          ▷ Decide sweeping rate
```

```
1  %1 =load p
2  store %1, q
3  %2 = volatile load q
4  if(isDangling(%2)):
5    OOT(&q)
6  else:
7    %3 = volatile load p
8    store %3, q
```

q = p

Figure 8: Prevent dangling pointer propagation. Code snippets with a dark background are instrumented by pSweeper.

instrument one check after the `store` instruction. In particular, we check whether `q` is dangling. If so, we nullify it.[3] This check is critical to prevent dangling pointer propagation. The reason is that although `q` has been swept before, `p` might have not been neutralized and the pointer assignment $q = p$ could propagate the dangling pointer from `p` to`q`. At first glance, the volatile `load` in Line 3 seems unnecessary and we can use `%1` directly. However, we must add this `load` instruction to prevent compiler and CPU from reordering the `isDangling()` check with the `store` instruction in Line 2. In other words, we must ensure that `isDanling(q)` check comes after the `store` instruction, so that the propagated dangling pointer `q` will be caught by the inlined `isDanling()` check. Note that this double-load strategy is not the only solution, and we can also intentionally introduce other data dependencies to prevent the reordering.

Then, if the `isDangling()` check in Line 4 fails, we reload the value of `p` and store it to `q`. There are two scenarios where the check in Line 4 can fail: (1) pointer `p` was not dangling (i.e., no risk of dangling pointer propagation at all) or (2) dangling pointer `p` has been neutralized and the freed memory is re-allocated. In the first scenario, the `store` in Line 8 is redundant but correct and safe. In the second scenario, `%3` must be different from `%1` and the dangling pointer propagation is successfully prevented. Note that, on multiprocessor systems, pSweeper uses CPU memory barriers like `mfence` to guarantee that the neutralized `p` is globally visible before the `isDangling()` check returns `false`.

We also need to insert `__asm__ __volatile__("":::"memory")` between the `load` instructions to prevent reordering by compilers. However, there is no need to insert memory barriers before `%3=load p`. This is because we only need to ensure that this `load` happens after the one in `isDangling(%2)`, regardless if `store` instructions have been globally visible before `%3=load p`.

Finally, we prove the correctness of this mechanism as follows:

- **Precondition.** Since we assume no concurrency bug, no one else except CPW thread will modify `p` or `q` during the code sequence in Figure 8.
- **Fact.** The race is harmful *iff* `q` is swept before `p`.
- **Completeness.** To prove the completeness of this mechanism, we only need to prove that, if both checks fail, `q` must NOT be dangling. We use proof by contradiction. **Proof:** Assume (`q` is dangling) $\Longrightarrow$ (`p` has not been neutralized before `%3=load p`) $\Longrightarrow$ (the pointed memory is still freed before `%3=load p`) $\Longrightarrow$ (`isDangling(%2)` must return true) $\Longrightarrow$ (`q` is set to NULL and `q` is not dangling). This contradicts the initial assumption.
- **Soundness.** We need to prove that, if either check succeeds, `q` must be dangling. The proof is straightforward based on the two preconditions.

4.6 More pSweeper Threads

pSweeper currently uses only one thread, which is sufficient in our evaluations. However, it can be extended to use multiple threads. The live pointers can be partitioned to segments, with each one being handled by one pSweeper thread during every round of sweeping. In this extension scheme, there is no race among pSweeper threads, and thus, no synchronization is required, making pSweeper quite scalable.

4.7 Object Origin Tracking (OOT)

It is notoriously difficult to analyze and locate bugs triggered in production runs [32, 37, 38, 53]. In order to facilitate the root-cause diagnosis of UaF vulnerabilities, pSweeper aims to provide not only

[3] Note that `isDangling()` checks if the pointer has already been neutralized.

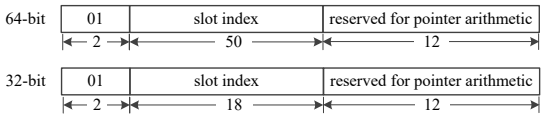

Figure 9: Use of pointer bits by OOT.

where dangling pointers are dereferenced (which can be obtained in core dumps) but also how objects are allocated and freed, i.e., object origin tracking (OOT). Unfortunately, it is non-trivial to link a dangling pointer access to the corresponding improper memory (de)allocation. Existing approaches like AddressSanitizer [51] and Exterminator [48] bind origin information with objects. However, this can cause inaccurate OOT when memory is reused, which is common in UaF exploits. Therefore, they are primarily suitable for in-house debugging but not for in-production diagnosis.

pSweeper instead encodes object origin information into dangling pointers. Such information is independent to memory reuse and can be propagated at no extra cost. The most significant two bits are set to `01` as in Figure 9 to ensure that applications crash safely upon dangling pointer dereference. Then, the origin information can be obtained in signal handlers. However, we must reserve sufficient least-significant bits to support pointer arithmetics. In our current implementation, we empirically reserve 12 bits.

OOT records the call stacks of `malloc()` and `free()` in a buffer slot that is assigned an index. The index is encoded into the middle 50 or 18 bits (with respect to 64-bit or 32-bit systems) during pointer neutralization, as shown in Figure 9. To reduce the memory overhead, the call stack information is compressed. In order to retrieve the slot index in `OOT`, pSweeper fills freed objects with corresponding slot indexes (Line 7 in Algorithm 5). In this way, given an in-bounds dangling pointer `p` to an object, pSweeper can easily construct the value to neutralize `p`.

When applications crash due to dangling pointer dereference, pSweeper extracts OOT information in signal handlers. However, Linux always returns zero, instead of the tagged pointer in Figure 9, as the illegal address in signal handlers. We address this by first obtaining the faulty instruction, e.g., `4008fe: movl %edx, (%rax)`, through `EIP/RIP` in signal handlers. This instruction informs that register `rax` contains the pointer value. Then, we can obtain the encoded origin information by reading that register.

Finally, the current design of OOT has two limitations. First, the encoded information may still be corrupted due to pointer arithmetics, even though 12 bits have been reserved. Fortunately, the reserved bits can handle most cases in practice. Second, in our current implementation, OOT is limited to record 2^{50} and 2^{18} objects that are live at the same time for 64-bit and 32-bit systems, respectively. However, such a recording capacity is sufficient for most software in practice. In particular, 64-bit systems have become prevalent nowadays and it is rare to create 2^{50} live objects at the same time.

5 EVALUATION

We implement a pSweeper prototype for x86-64, on top of LLVM 3.7 compiler infrastructure [10, 35], and use LLVM's link-time optimization support (LTO) for the whole program analysis. The static analysis and instrumentation pass in pSweeper operates on LLVM intermediate representation (IR). Our current prototype employs some preliminary optimizations, e.g., inlining operations in Algorithm 2 and Figure 8 when instrumenting `store` instructions to avoid function calls.

We evaluate pSweeper by answering four questions:

- Is pSweeper effective to mitigate real UaF vulnerabilities?
- What is the performance overhead of pSweeper?
- How scalable is pSweeper for multi-threaded applications?
- Can pSweeper efficiently work on complex software?

All experiments are conducted on 64-bit Ubuntu-16.04 with a 2-core 2-thread (i.e., 4 threads in total) Intel(R) Core(TM) i5-4300U at 1.9GHz with 12GB RAM.

5.1 Effectiveness of pSweeper

To evaluate the effectiveness of pSweeper, we apply it to four real-world UaF vulnerabilities in three applications, as listed in Table 2. pSweeper successfully neutralizes the unsafe dangling pointers and pinpoints the root-causes in all four cases. Due to space limit, we next describe CVE-2016-6309 only in details.

CVE/Bug ID	Application	Protected
CVE-2016-6309 [4]	OpenSSL 1.1.0a	✔
CVE-2014-3505 [3]	OpenSSL <1.01i	✔
Bug 12840 [13]	Wireshark	✔
Bug 2440 [9]	Lighttpd 1.4.32	✔

Table 2: Real-world UaF vulnerabilities used for evaluation.

CVE-2016-6309 in OpenSSL is caused by memory reallocation in `statem.c:548`. OpenSSL initially allocates a buffer of 16KB to receive messages. When a larger message is received, the buffer is reallocated using `CRYPTO_clear_realloc()`, which essentially allocates a new buffer and frees the old one. Therefore, the underlying location of the buffer is changed. However, a pointer `s→init_msg` is not updated and still refers to the old location.

When this vulnerability is exploited, there can be two cases. First, due to deferred free and asynchronous neutralization, if the dangling is accessed before being neutralized, the openSSL server can always execute normally. On the other hand, if it is exploited after neutralization, the openSSL server crashes safely and pSweeper successfully pinpoints `OPENSSL_clear_realloc()` in `BUF_MEM_grow_clean()` (`buffer.c:109`) as root cause.

5.2 Performance on SPEC CPU2006

We next evaluate the performance overhead of pSweeper on SPEC CPU2006 benchmarks. Table 3 presents the statistical results of SPEC CPU2006 benchmarks when pSweeper runs at a sweeping rate of one second.

As can be seen, pSweeper finds similar number of pointers (Column 5 in Table 3) as DangSan, which is far more than DANGNULL. This demonstrates that pSweeper has comparative coverage to the state-of-the-art defense systems. We also find that pSweeper neutralizes fewer pointers (Column 7 in Table 3) than DangSan, although more than DANGNULL. This is because pSweeper concurrently sweeps dangling pointers in a dedicated thread and does not stall applications. As a result, although a pointer is dangling

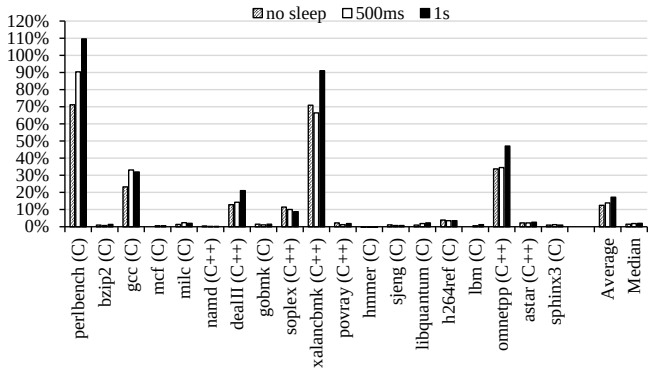

Figure 10: pSweeper's performance on SPEC CPU2006.

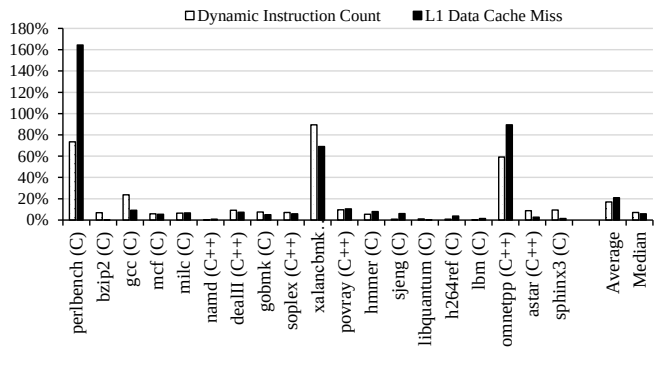

Figure 11: Dynamic instruction overhead and L1 data cache misses on SPEC CPU2006.

at `free()`, it probably has been overwritten by applications with non-dangling values when pSweeper checks it. In particular, the majority of dangling pointers identified in DangSan are on stack [54], which become invalid after function returns. Also, it is possible that the objects containing dangling pointers have been freed before pSweeper starts to sweep. Most of these invalid dangling pointers are ignored by pSweeper. We emphasize that neutralizing these stale dangling pointers does not increase the security guarantee and pSweeper provides the same protection as previous systems.

5.2.1 Runtime Overhead.

Figure 10 presents the performance overhead of pSweeper at different sweeping rates, i.e., no sleep, 500ms sleep, and 1s sleep between sweeping rounds. The overhead is normalized over the baseline and all the results are averaged over three consecutive runs. The average overheads of pSweeper at different sweeping rates are 12.5% (no sleep), 13.9% (500ms), and 17.2% (1s).

Effect of sweeping rate. Generally, sweeping rates do not significantly affect the performance of applications as pSweeper concurrently runs on spare cores. Therefore, we an see that all three configurations (1s, 500ms, and nosleep) induce similar and trivial overhead on most benchmark. However, we find that the benchmarks like `perlbench`, `gcc`, `omnetpp`, and `xalancbmk` still suffer high overhead. In particular, the overhead of these benchmarks basically positively correlates with sweeping intervals, i.e., the larger the interval, the larger the overhead. There are mainly two reasons. On the one hand, these benchmarks are memory allocation intensive. Simply intercepting and maintaining metadata in pSweeper can incur a large overhead. On the other hand, when pSweeper runs at a larger interval, memory free requests are deferred for a longer time. An allocation-intensive application like `gcc` may not be able to immediately reuse the freed memory. As a result, much more time is spent in kernel mode when memory allocators try to allocate new objects.

Static instrumentation overhead. We now break down the overhead caused by static code instrumentation. The overhead mainly comes from the hooked `malloc()` family of functions, which set up object metadata, maintain live objects and MAS table. They introduce a bunch of extra memory writes for each allocated object. We find that they account for about 5.6% of the average overhead. Especially, in the case of allocation-intensive applications, the accumulated overhead is high, e.g., ~15% for `gcc`. Finally, the instrumented `store` instruction, which is the main performance bottleneck in previous works [36, 54, 58], causes low overhead in pSweeper, about 1.8%.

Dynamic instruction count and data cache overhead. We use hardware performance counters to measure the dynamic instruction counts and cache misses of the 32KB L1 data cache. We are only interested in the overhead caused by the instrumented code. Thus, we do not spawn the pSweeper thread and disable deferred free. The results are plotted in Figure 11, showing that dynamic instruction counts highly correlate with the runtime overhead and are the main source of overheads for most benchmarks. A noticeable negative impact of MAS and PLM tables is the additional data cache misses, resulting in a large portion of performance overhead.

5.2.2 Memory Overhead.

Figure 12 shows that pSweeper moderately increases memory footprint in terms of maximum resident set size, with average overheads 112.5% (no sleep), 169.7% (500ms), and 247.3% (1s).

Generally, faster sweeping rates result in lower memory overhead. This is because free requests are deferred shorter, and thus memory can be freed faster. A faster sweeping rate is especially important to allocation-intensive applications. For instance, sweeping at 500ms, compared to 1s, reduces the memory overhead of `gcc` by an order of magnitude. Other sources of memory overhead include MAS table, `ObjList`, `PtrList` and `PLM` table. We can see that these metadata consumes acceptable amount of memory. In particular, since several benchmarks allocate a small number of large objects (Column 4 Table 3), the compression strategy used in MAS table can greatly reduce memory overhead.

We also find that a large portion of memory overhead can be attributed to a single benchmark `dealII`, which incurs about 1150% memory overhead. The exceptionally high overhead is caused by two factors. On the one hand, the baseline of `dealII` has a small memory footprint and thus the relative memory consumption of pSweeper's metadata becomes high. On the other hand, the deferred frees make pSweeper have a even larger overhead relative to the baseline. Excluding `dealII`, the memory overhead can drop to 54.9% for pSweeper-nosleep, 81% for pSweeper-500ms, and 144.3% for pSweeper-1s.

Benchmark	# of Allocations	# of Frees	Avg. Object Size (Bytes)	Total # of Pointers	Peak # of Pointers	# of Pointers Neutralized
perlbench (C)	358M†	356M	514	40,490M	971,353	421,352
bzip2 (C)	7,182	4,440	1.6M	2.2M	1,184	0
gcc (C)	28M	28M	26,088	7,170M	438,016	186,451
mcf (C)	1,174	721	1.4M	7,658M	173,625	0
milc (C)	7,686	7,184	11M	2,585M	76,254	0
namd (C++)	2,493	2,038	19,582	2.9M	1,746	0
gobmk (C)	663,879	658,695	1,707	607M	28,841	86
dealII (C++)	151M	151M	82	117M	329,194	7,293
soplex (C++)	312,951	310,613	189,852	836M	76,278	553
povray (C++)	2.4M	2.4M	56	4,679M	128,525	7,428
hmmer (C)	2.4M	2.4M	1,048	3.8M	2,237	0
sjeng (C)	1,174	717	154,809	3	0	0
libquantum (C)	1,348	895	1.1M	186	8	0
h264ref (C)	182,784	181,283	7,735	11M	3,674	961
omnetpp (C++)	267M	266M	174	13,099M	589,145	13,152
lbm (C)	1,173	720	367,047	5,949	28	0
astar (C++)	4.8M	4.8M	922	1,235M	34,519	104
xalancbmk (C++)	135M	135M	466	2,387M	418,924	78,618
sphinx3 (C)	14M	14M	1,136	302M	24,923	762

Table 3: Detailed results on SPEC CPU2006 benchmarks. pSweeper runs at 1s sweeping rate. †M for million.

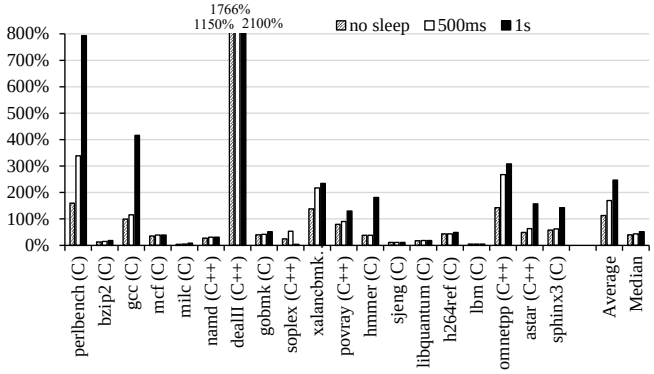

Figure 12: Memory overhead on SPEC CPU2006.

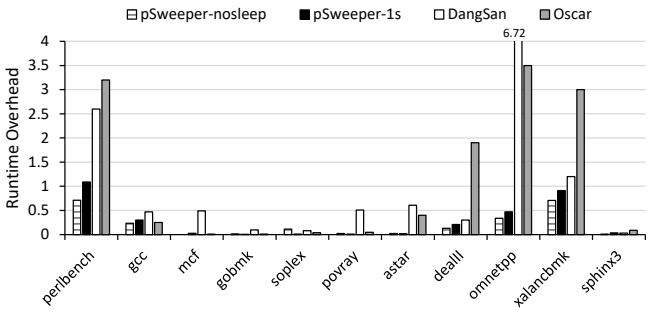

Figure 13: Overhead comparison with DangSan and Oscar.

5.2.3 Comparison to DangSan and Oscar.

We compare pSweeper with DangSan [54] and Oscar [24], two state-of-the-art UaF defense systems with best performance. We use two sweeping rates of pSweeper, 1s and nosleep. The geometric mean of DangSan is a 41% slowdown and Oscar is 40%, while pSweeper-nosleep is 12.5% and pSweeper-1s is 17.2%. Figure 13 compares the eleven benchmarks on which pSweeper obviously outperforms DangSan and Oscar. There is no remarkable difference on other benchmarks. In terms of memory overhead, DangSan imposes an average overhead of 210%, while pSweeper-nosleep is 112.5% and pSweeper-1s is 247.3%. Oscar has 52% memory overhead, which is more efficient than both DangSan and pSweeper. Since the memory overhead of pSpweer is mainly due to deferred free, our future direction is to design a memory efficient allocator for deferred free. pSweeper achieves much lower slowdown in application performance mainly because the heavyweight security check and enforcement have been outsourced to dedicated threads that can leverage the spare cores in modern systems. In particular, the workload induced to application threads is quite light.

5.3 Scalability on Multi-threaded Applications

We use PARSEC 3.0 [17] to evaluate the scalability of pSweeper with respect to an increasing number of application threads. Our baseline LLVM fails to compile four benchmarks and Figure 14 shows the results for nine succeeded ones. As we can see, pSweeper scales nearly as well as the baseline on all benchmarks. This is mainly because lock-free algorithms are devised to address almost all races between pSweeper and application threads. As a result, the incurred overhead does not increase significantly when systems become more contended. For pSweeper-nosleep, the core running the

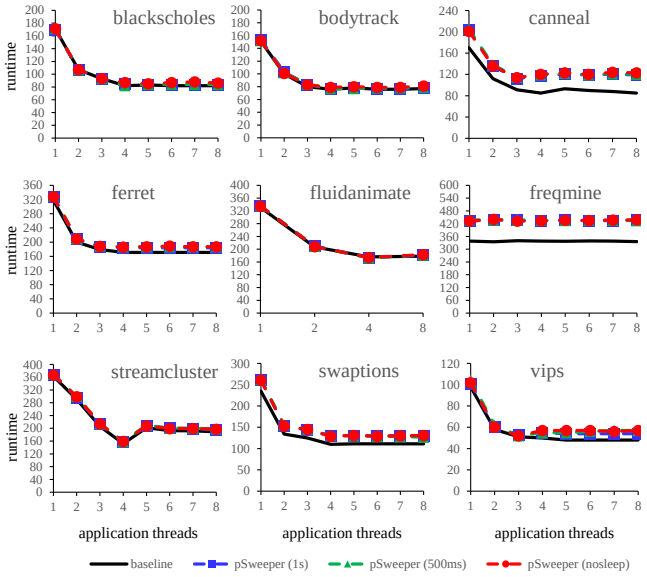

Figure 14: Scalability of pSweeper on PARSEC 3.0. The number of threads must be a power of two for `fluidanimate`.

pSweeper thread always has a utilization of 100%. For most benchmarks, pSweeper consumes about 20% and less than 1% CPU under 500ms and 1s sweeping rates, respectively. The CPU utilization increases to 90% (500ms) and 10~30% (1s) for memory-allocation intensive benchmarks, such as gcc, perlbench, and xalancbmk. We also observe that the runtime of benchmarks does not decrease anymore after the number of application threads reaches four. This is because our CPU has only two cores with 2-thread hyperthreading on each core (i.e., providing 4-thread hyperthreading in total). The geometric means of overhead over all nine benchmarks range from 7.5% to 18.1% for all three configurations of pSweeper. We find that, when the number of application threads grows larger than the number of CPU threads (four in our case), pSweeper-nosleep incurs relatively larger overhead than pSweeper-1s because pSweeper-nosleep exacerbates the CPU contention, while pSweeper-1s consumes relatively lower CPU resources. The memory overhead basically does not highly correlate with the number of application threads and the geometric means of overheads are 1600% (1s), 840% (500ms), and 49% (no sleep), respectively. The high overhead is mostly due to `swaptions` that consumes 144x memory for pSweeper (1s). The reason is that the memory footprint of baseline `swaptions` is quite small. As a result, the memory consumption caused by pSweeper becomes exceptionally large relative to the baseline. Excluding `swaptions` which is not evaluated by DangSan, the memory overheads of pSweeper become 27% (1s), 22% (500ms), and 18% (no sleep), respectively.

5.4 Macro Benchmarks

We now demonstrate that pSweeper works efficiently on modern applications with two case studies, Lighttpd web server and Firefox browser.

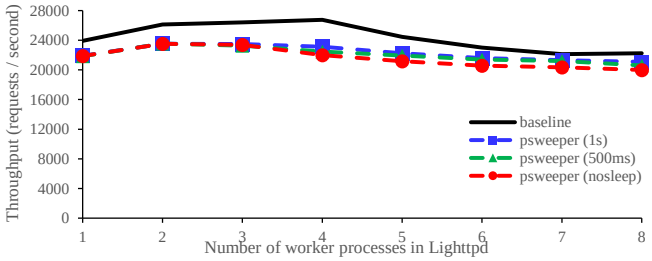

Figure 15: pSweeper overhead on Lighttpd.

5.4.1 *Lighttpd.*

We first conduct experiments on Lighttpd 1.4.40. To generate client requests, we run the ApacheBench [1] tool on a second desktop. The tool makes 100,000 requests with 128 concurrent connections to transfer a 50-byte file. We use a very small file to minimize the potential variance caused by network and disk I/O. The results are averaged over five runs.

Figure 15 shows the throughput of Lighttpd with respect to the different number of worker processes. We can see that pSweeper scales well on Lighttpd with overheads in ranges of 3.4%~17.8%. The highest overhead occurs when the number of worker processes reach four which occupy all four hyperthreading in two cores. When the number of worker processes become larger than four, the baseline throughput also decreases because application processes now contend with each other. This result is consistent with those in §5.3. The memory overheads are about 158.7% (1s), 42.2% (500ms), and 38.7% (no sleep) in all cases.

5.4.2 *Mozilla Firefox.*

We choose Firefox 47.0 as our second case study. Table 4 presents the evaluation results on three popular browser benchmarks, MotionMark [11] assessing a browser's capability to animate complex scenes at a target frame rate, Speedometer [12] measuring simulated user interactions in web applications, and JetStream 1.1 [8] covering a variety of advanced Javascript workloads. In all three benchmarks, larger scores indicate higher performance. We can see that the induced runtime overhead is quite low, ranging from 2.3% to 8.1%. The geometric means of memory overhead are 863%, 374%, and 117%, for pSweeper-1s, -500ms, and -nosleep, respectively.

We further evaluate pSweeper by visiting Alexa Top 50 websites. We encounter no error during the test of accessing the websites. Table 5 lists the page load time (using app.telemetry [2]) when visiting five popular websites. On average, the page load time increases by 3.7%.

Benchmarks	MotionMark	Speedometer	JetStream 1.1
	Score	Runs /minute	Score
Baseline	145.24	48.5	160.63
1s	133.98 (7.7%)	44.4 (6.2%)	156.9 (2.3%)
500ms	133.49 (8.1%)	45.12 (6.9%)	156.6 (2.5%)
nosleep	134.13 (7.6%)	44.7 (7.8%)	156.1 (2.8%)

Table 4: Overhead of pSweeper(-1s, -500ms, -nosleep) on three browser benchmarks. The percentage in parentheses is the slowdown.

Websites	Baseline	pSweeper		
		1s	500ms	nosleep
google.com	0.55	0.57	0.56	0.58
youtube.com	1.95	1.99	2.02	2.01
facebook.com	0.64	0.66	0.66	0.67
amazon.com	2.42	2.51	2.52	2.50
yahoo.com	3.51	3.61	3.63	3.59

Table 5: Page load time (in seconds) of pSweeper on five popular websites.

5.5 Summary

In summary, faster sweeping rates can generally have lower runtime and memory overheads. However, pSweeper with faster sweeping rates will consume more energy and CPU resources. While this is appropriate when there are idle cores, it may seriously affect the performance of CPU-bound multi-threaded applications. Therefore, if an application is not memory-allocation-intensive, pSweeper should be usually configured to run at a lower sweeping rate.

6 DISCUSSION & LIMITATIONS

Comparison to garbage collection (GC). GC [6, 18] not only heads off exploits but also prevents program crashes due to UaF vulnerabilities. However, most GC algorithms consume more memory because they defer free until there is insufficient memory or applications explicitly ask for. Even worse, some dangling pointers can remain alive for a long time, thus preventing conservative GC reclaiming freed memory. By contrast, pSweeper frees memory after one round of pointer sweeping and can proactively eliminate dangling pointers. Moreover, stop-the-world GC can cause unpredictable interference to application performance. pSweeper instead does not stall application threads. Finally, although pSweeper can only probabilistically mask program crashes, it guarantees to pinpoint the root-causes of UaF vulnerabilities when programs crash.

pSweeper metadata protection. pSweeper does not specially protect its metadata like the MAS and PLM table. However, this does not degrade our security guarantee. By design, all UaF exploits are disrupted. Thus, attackers can leak and tamper with metadata only through non-UaF vulnerabilities. As discussed in §2, orthogonal defenses should be used to protect against these vulnerabilities.

Accessing freed memory due to deferred free. Since pSweeper defers object free until the end of a sweeping round, applications are able to access the memory that should have been freed. This design resembles garbage collection. Therefore, we believe this is not a critical concern in practice.

Energy consumption. Since pSweeper continuously scans for dangling pointers in a concurrent thread, it will consume more power and energy. As a result, it may not be suitable for deployment on battery-backed mobile devices. Instead, we envision pSweeper to be mainly deployed on desktops.

False positives. Basically, false positives can occur in two cases. First, a pointer may be type-casted to and used as an integer. For instance, a program might depend on the difference of two pointers p, q. If p or q is neutralized by pSweeper, the value $(p - q)$ will be changed. Second, applications may intentionally use the values in dangling pointers. Since these false positives are rare in practice, we believe they will not seriously affect the practicality of pSweeper. Actually, all other three comparable approaches [36, 54, 58] suffer the same false positives.

False negatives. pSweeper relies on the types of global/local variables and operands in `store` instructions to identify live pointers. However, an integer is type-casted to a pointer at runtime. Also, pSweeper currently conservatively ignores unions if one of their fields are non-pointers. In these cases, pSweeper will suffer false negatives if the missed pointers become dangling.

Another possible cause of false negatives lies in the fact that pSweeper does not proactively neutralize dangling pointers in registers. It will induce undue overhead if pSweeper peeks into and tampers with the registers used by application threads. While these dangling pointers are theoretically false negatives, they can hardly be exploited in practice. Therefore, currently we do not tackle them. Instead, we guarantee that they never propagate to memory (§4.5). Again, all existing approaches [36, 54, 58] do no handle dangling pointer in registers.

7 RELATED WORK

We have compared pSweeper with DANGNULL [36], FreeSentry [58] and DangSan [54], the works closest to ours above (§3.1). Here we discuss the remaining related works.

Dangling pointer detection. Tools like Valgrind [45] and AddressSanitizer [51] track the (de)allocation status of each memory location. As long as a freed memory block is not reallocated, these tools can detect all dangling pointers. However, they can miss those pointing to a reallocated memory, which is common in UaF exploits. Another set of approaches extend each pointer with a unique identifier and check the validity on every pointer dereference [15, 43, 56, 57]. Unfortunately, software-only explicit pointer checks can slow applications by an order of magnitude. Recently, Nagarakatte et al. [40, 41] proposed a hardware-assisted approach that can provide full memory safety at low overheads. Undangle [19] detects dangling pointers by using dynamic taint analysis to track pointer propagations at runtime. It can serve as an in-house testing tool but not a runtime defense system.

Safe memory allocators. Cling [14] is a safe memory allocator that avoids memory reuse among objects of different types. It can thwart many, but not all, UaF exploits. DieHard [16] and DieHarder [47] are based on the idea of “infinite" heaps. Unfortunately, an infinite-heap is idealized but infeasible, and thus it can only provide probabilistic memory safety. Exterminator [48] extends DieHard to automatically fix dangling pointers by delaying object frees. Dhurjati and Adve [26] used a new virtual page for each memory allocation and relied on page protection to detect dangling pointer accesses. Inspired by Dhurjati and Adve's work, Oscar [24] develops a page-permission-based protection scheme to ensure pointer safety. By contrast, pSweeper proactively neutralizes all dangling pointers.

Safe C languages. Fail-safe C [49] implements a completely memory-safe compiler that is fully compatible with ANSI C. It uses garbage collection to protect against dangling pointers. There are also safe C dialects, such as Cyclone [28, 30] and CCured [21, 44]. Although they attempt to keep compatible with C/C++ specifications, non-trivial efforts are still needed to retrofit legacy programs.

Parallelizing security checks. Concurrent security checks as in pSweeper have also been adopted in several previous works.

Speck [46] decouples security checks from applications and executes them in parallel on multiple cores. Unlike Speck, pSweeper does not use speculative execution. Cruiser [59] and Kruiser [52] use concurrent threads to detect buffer overflows in user applications and kernels, respectively. ShadowReplica [29] accelerates dynamic data flow tracking by running analysis on spare cores. However, pSweeper tackles a different problem and faces unique challenges. Finally, RCORE [27] detects program state invariant violations on idle cores. Although RCORE can also detect dangling pointers, it does not consider the race conditions (e.g., dangling pointer propagation §4.5 and no deferred free in RCORE). RCORE also relies on static type analysis to identify pointers, which is quite challenging to be complete in real-world software. Therefore, pSweeper is more robust than RCORE.

8 CONCLUSION

This paper presents pSweeper, a system that effectively protects applications from UaF vulnerabilities at low overhead. The key feature of pSweeper is to iteratively sweep live pointers to neutralize dangling ones in concurrent threads. To accomplish this, we devise lock-free algorithms to address the entangled races among pSweeper and application threads, without using any heavyweight synchronization mechanism that can stall application threads. We also propose to encode object origin information into dangling pointers to achieve object origin tracking, which helps to pinpoint the root-causes of UaF vulnerabilities. We implement a prototype of pSweeper and validate its effectiveness and efficiency in production environments.

9 ACKNOWLEDGMENTS

We would like to thank our shepherd Byoungyoung Lee and anonymous reviewers for their insightful feedback, which helped us improve the quality of this paper. This work was supported in part by ONR grant N00014-17-1-248.

REFERENCES

[1] Apachebench. https://httpd.apache.org/docs/2.4/programs/ab.html/.
[2] app.telemetry page speed monitor. https://addons.mozilla.org/en-US/firefox/addon/apptelemetry/.
[3] Cve-2014-3505. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3505.
[4] Cve-2016-6309. https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2016-6309.
[5] Detecting memory access errors using hardware support. https://blogs.oracle.com/raj/entry/detecting_memory_access_errors_using.
[6] A garbage collector for c and c++. http://hboehm.info/gc/.
[7] Intel 64 and ia-32 architectures software developer manual. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html.
[8] Jetstream. http://browserbench.org/JetStream/.
[9] Lighttpd bug-2440. https://redmine.lighttpd.net/issues/2440.
[10] The llvm compiler infrastructure. http://llvm.org/.
[11] Motionmark. http://browserbench.org/MotionMark/.
[12] Speedometer. http://browserbench.org/Speedometer/.
[13] Wireshark bug-12840. https://bugs.wireshark.org/bugzilla/show_bug.cgi?id=12840.
[14] Akritidis, P. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX Conference on Security (Security)* (2010).
[15] Austin, T. M., Breach, S. E., and Sohi, G. S. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI)* (1994).
[16] Berger, E. D., and Zorn, B. G. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2006).
[17] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2008).
[18] Boehm, H.-J., and Weiser, M. Garbage collection in an uncooperative environment. *Softw. Pract. Exper. 18* (1988).
[19] Caballero, J., Grieco, G., Marron, M., and Nappa, A. 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 (ISSTA)* (2012).
[20] Chen, X., Slowinska, A., Andriesse, D., Bos, H., and Giuffrida, C. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *Proc. Network and Distributed System Security Symposium (NDSS)* (2015).
[21] Condit, J., Harren, M., McPeak, S., Necula, G. C., and Weimer, W. Ccured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)* (2003).
[22] Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium* (1998).
[23] Criswell, J., Lenharth, A., Dhurjati, D., and Adve, V. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2007).
[24] Dang, T. H., Maniatis, P., and Wagner, D. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *26th USENIX Security Symposium* (2017).
[25] Devietti, J., Blundell, C., Martin, M. M. K., and Zdancewic, S. Hardbound: Architectural support for spatial safety of the c programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2008).
[26] Dhurjati, D., and Adve, V. Efficiently detecting all dangling pointer uses in production servers. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)* (2006).
[27] Giuffrida, C., Cavallaro, L., and Tanenbaum, A. S. Practical automated vulnerability monitoring using program state invariants. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2013).
[28] Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., and Cheney, J. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)* (2002).
[29] Jee, K., Kemerlis, V. P., Keromytis, A. D., and Portokalidis, G. Shadowreplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2013).
[30] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y. Cyclone: A safe dialect of c. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATC)* (2002).
[31] Jones, R., Hosking, A., and Moss, E. *The Garbage Collection Handbook: The Art of Automatic Memory Management*, 1st ed. Chapman & Hall/CRC, 2011.
[32] Kasikci, B., Schubert, B., Pereira, C., Pokam, G., and Candea, G. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)* (2015).
[33] Kharbutli, M., Jiang, X., Solihin, Y., Venkataramani, G., and Prvulovic, M. Comprehensively and efficiently protecting the heap. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2006).
[34] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2014).
[35] Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO)* (2004).
[36] Lee, B., Song, C., Jang, Y., Wang, T., Kim, T., Lu, L., and Lee, W. Preventing use-after-free with dangling pointers nullification. In *Proc. Network and Distributed System Security Symposium (NDSS)* (2015).
[37] Liblit, B., Aiken, A., Zheng, A. X., and Jordan, M. I. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)* (2003).
[38] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2005).
[39] Lvin, V. B., Novark, G., Berger, E. D., and Zorn, B. G. Archipelago: Trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2008).

[40] Nagarakatte, S., Martin, M. M. K., and Zdancewic, S. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)* (2012).

[41] Nagarakatte, S., Martin, M. M. K., and Zdancewic, S. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2014).

[42] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2009).

[43] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM)* (2010).

[44] Necula, G. C., McPeak, S., and Weimer, W. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2002).

[45] Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2007).

[46] Nightingale, E. B., Peek, D., Chen, P. M., and Flinn, J. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2008).

[47] Novark, G., and Berger, E. D. Dieharder: Securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)* (2010).

[48] Novark, G., Berger, E. D., and Zorn, B. G. Exterminator: Automatically correcting memory errors with high probability. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2007).

[49] Oiwa, Y. Implementation of the memory-safe full ansi-c compiler. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2009).

[50] Prasad, M., and cker Chiueh, T. A binary rewriting defense against stack based overflow attacks. In *In Proceedings of the USENIX Annual Technical Conference (ATC)* (2003).

[51] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (ATC)* (2012).

[52] Tian, D., Zeng, Q., Wu, D., Liu, P., and Hu, C. Kruiser: Semi-synchronized non-blocking concurrent kernel heap buffer overflow monitoring. In *Proc. Network and Distributed System Security Symposium (NDSS)* (2015).

[53] Tucek, J., Lu, S., Huang, C., Xanthos, S., and Zhou, Y. Triage: Diagnosing production run failures at the user's site. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2007).

[54] van der Kouwe, E., Nigade, V., and Giuffrida, C. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)* (2017).

[55] Venkataramani, G., Roemer, B., Solihin, Y., and Prvulovic, M. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)* (2007).

[56] Xu, W., DuVarney, D. C., and Sekar, R. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (FSE)* (2004).

[57] Yong, S. H., and Horwitz, S. Protecting c programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (2003).

[58] Younan, Y. Freesentry: Protecting against use-after-free vulnerabilities due to dangling pointers. In *Proc. Network and Distributed System Security Symposium (NDSS)* (2015).

[59] Zeng, Q., Wu, D., and Liu, P. Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2011).