Sampler: PMU-based Sampling to Detect Memory Errors Latent in Production Software

Sam Silvestro* Hongyu Liu* Tong Zhang† Changhee Jung† Dongyoon Lee† Tongping Liu*
*University of Texas at San Antonio
†Virginia Tech

***Abstract*—Deployed software is still faced with numerous in-production memory errors. They can significantly affect system reliability and security, causing application crashes, erratic execution behavior, or security attacks. Unfortunately, existing tools cannot be deployed in the production environment, since they either impose significant performance/memory overhead, or can only detect partial errors. This paper presents `Sampler`, a library that employs the combination of hardware-based SAMPLing and novel heap allocator design to efficiently identify a range of memory ERrors, including buffer overflows, use-after-frees, invalid frees, and double-frees. Due to the stringent Quality of Service (QoS) requirement of production services, `Sampler` proposes to trade detection effectiveness for performance on each execution. Rather than inspecting every memory access, `Sampler` proposes the use of the Performance Monitoring Unit (PMU) hardware to sample memory accesses, and only checks the validity of sampled accesses. At the same time, `Sampler` proposes a novel dynamic allocator supporting fast metadata lookup, and a solution to prevent false alarms potentially caused by sampling. The sampling-based approach, although it may lead to reduced effectiveness on each execution, is suitable for in-production software, since software is generally employed by a large number of individuals, and may be executed many times or over a long period of time. By randomizing the start of the sampling, different executions may sample different sequences of memory accesses, working together to enable effective detection. Experimental results demonstrate that `Sampler` detects all known memory bugs inside real applications, without any false positive. `Sampler` only imposes negligible performance overhead (2.4% on average). `Sampler` is the first work that simultaneously satisfies efficiency, preciseness, completeness, accuracy, and transparency, making it a practical tool for in-production deployment.**

***Index Terms*—Sampling, PMU, Vulnerability Detection**

I. Introduction

Memory errors, such as buffer overflows and use-after-frees, have plagued deployed C/C++ software for decades [1], [2], causing programs to crash or produce incorrect results. Even worse, they can be exploited to launch security attacks, resulting in the leakage of private data, and even the hijacking of the whole machine [1].

It is impossible to expunge all memory errors during development phases via static analysis and dynamic testing. Static analysis approaches have numerous false positives and/or scalability issues [1]. Although dynamic testing overcomes these two issues, it requires specific inputs, timing, or susceptible schedules (especially for multithreaded programs) to expose the bugs [3], which can never be achieved given insufficient testing time and imperfect testing environments [4]. Therefore, numerous memory errors are inevitably leaked to the deployed environment. As shown in TABLE I, a significant number of memory errors hidden in deployed software were reported in a single year by the NVD database [5].

Memory Errors	Heap Overflow	Heap Over-read	Invalid-free	Double-free	Use-after-free
Occurrences (#)	673	125	35	33	264

TABLE I
Reported heap errors in the past year.

Unfortunately, none of the existing dynamic tools have been employed in the production environment due to their serious shortcomings. Many tools incur significant performance overhead [2], [6]–[15]. For instance, Valgrind introduces more than $20\times$ overhead [9], [16], caused by its expensive dynamic instrumentation and checking prior to every memory access. AddressSanitizer, a popular work from Google Inc., employs static analysis to reduce the checking of memory accesses, but still imposes over 70% performance overhead, and more than $3\times$ memory overhead [2]. More importantly, AddressSanitizer can only detect errors of instrumented components, thereby missing errors caused by any third-party or non-instrumented libraries [17].

Although lightweight techniques exist [16], [18]–[20], they also have their own serious limitations. SafeMem functions only on machines equipped with Error-Correcting Code (ECC) memory (typically only available on high-end servers), and requires changes to the underlying operating system [18]. Due to the re-purposing of ECC, SafeMem unavoidably compromises the original goal of ECC, i.e., preventing data corruption. Cruiser only reports the possibilities of buffer overflows, but cannot inform the specific locations of these errors [19]. DoubleTake [16] and iReplayer [20], two recent detection tools, can only detect write-based memory errors, while overlooking a larger portion of read-based errors.

This paper proposes `Sampler`, a tool that simultaneously satisfies all of the key properties necessary to be employed in the production environment.

- **Efficiency:** The detection overhead should be extremely low, due to the stringent Quality of Service (QoS) demand of in-production software, e.g., $< 5\%$ [21], [22].
- **Preciseness:** The report should include sufficient information to assist programmers with fixing the bugs.
- **Completeness**: The tool should be able to detect both read and write errors caused by all components.
- **Accuracy:** Every reported error should be a real problem. In fact, it is extremely difficult for programmers to con-

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00027

firm in-production errors due to the lack of runtime environment, such as sensitive inputs, third-party libraries, or execution records [23], [24].

- **Transparency:** The detection procedure should be transparent to normal users, requiring zero effort from them, since they may not have the expertise or willingness to perform any additional tasks.

We observe that the deployed software is generally utilized by a large number of individuals, e.g. over 1 billion Microsoft Office users [25], or is executed many times or over a long period of time. This fact makes it possible to crowdsource the error detection [26]: *for each individual execution, `Sampler` only incurs a minimum runtime overhead to detect partial errors; however, we could employ myriad executions together to detect memory errors.*

`Sampler` proposes to utilize the sampling approach in order to reduce its individual detection overhead. More specifically, `Sampler` leverages the ubiquitous and off-the-shelf Performance Monitoring Unit (PMU) hardware to trace/sample memory accesses at a very low cost. PMU-based sampling is non-intrusive, since it requires neither changes of applications nor explicit instrumentation. Unlike existing work [2], [16], the PMU hardware is able to sample memory reads and writes of all components, thus detecting errors hidden in any component. In addition, sampled events include precise instruction pointers (IP) that can provide the line-of-code reference of the errors. Therefore, PMU-based sampling has the potential to satisfy the efficiency, completeness, and preciseness properties, but in reality there are multiple technical challenges, listed as follows.

First, many factors may still lead to significant performance overhead, even with PMU-based sampling: (1) the overhead can be prohibitively high if we must perform checking upon every sample, due to the overhead caused by frequent interrupts. Instead, `Sampler` configures the kernel to notice the collection of samples when the buffer is full, and further overcomes possible correctness issues by integrating with its custom memory allocator. (2) The PMU hardware may generate an excessive number of samples, e.g. thousands of samples each second, even with a sampling period of 5000, implying significant pressure for fast error checking. To tackle this problem, `Sampler` designs a novel memory allocator that provides an **information-computable** capability: given any sampled address inside the heap, `Sampler` can compute the starting address, the size class, and the metadata placement of this object. We also further design the special layout of the allocator to accelerate these computations upon every checking operation.

Second, false positives or incorrect reports may be generated when using the sampling mechanism. (1) Existing detectors embed the metadata between actual heap objects [2], [16], [20], which can be easily corrupted by memory errors, such as buffer overflows or use-after-frees. Under these circumstances, false alarms can be generated. To avoid such corruptions, `Sampler` designs a BIBOP-style (meaning “big bag of pages” [27]) allocator that separates the metadata from actual objects. (2) It is challenging to determine the temporal relationship between memory references and free operations. To avoid false reports caused by an incorrect order, `Sampler` employs the precise hardware timestamp of modern machines to mark the events, and coordinates the actual free operations of its custom allocator. Combining the detection with its custom allocator provides other benefits as well, e.g. avoiding the memory consumption caused by a bitmap, as further described in Section III.

Third, the effectiveness can be greatly limited by the sampling frequency. It is impractical to significantly increase the sampling frequency due to potential high performance overhead. `Sampler` instead proposes to randomly initiate the sampling for different threads across different executions, while keeping a stable overhead. This randomization mechanism is able to coordinate different executions altogether (without using a centralized mechanism) to detect errors latent in production software. This is because different executions may experience different sequences of memory references, thus detecting different errors.

Contributions

Overall, this paper makes the following contributions:

a) The first work utilizing PMU-based sampling for detecting memory errors

`Sampler` is the first to propose the use of sampling to detect memory errors, and is also the first work that leverages the Performance Monitoring Unit (PMU) hardware for detecting these errors. It further proposes to cooperate with its custom allocator to avoid false alarms and reduce detection overhead.

b) A faster memory allocator with the “information-computable” property

`Sampler` designs a novel BIBOP-style memory allocator that provides the information-computable capability, by taking advantage of the vast address space of 64-bit machines. The allocator runs around 3% faster than the standard Linux allocator, and enables the fast lookup of metadata. The allocator is also safer than the Linux allocator, by separating its metadata from the actual heap. However, it imposes around 34% memory overhead.

c) A practical tool that can be transparently utilized in the production environment

`Sampler` is a dynamic library that can simply be linked to applications, which requires no change to the underlying OS, no recompilation of legacy software, and no change of applications. `Sampler` requires zero manual effort from users. Experimental results show that `Sampler` detects known bugs in widely-used applications, while imposing negligible performance overhead ($\approx$ 2.4% on average).

Paper Outline

The remainder of this paper is organized as follows. First, Section II briefly discusses the background of the PMU, as well as ideas for detecting different memory errors. Section III presents the design of `Sampler`’s memory allocator, and Section IV presents the detailed implementation of `Sampler`.

Then, Section V discusses the effectiveness and limitations of `Sampler`. Section VI further evaluates the effectiveness, performance overhead, and memory overhead of `Sampler`. In the end, Section VII discusses related works, while Section VIII concludes the paper.

II. Overview

This section explains the background of PMU-based sampling, technical challenges, and the basic ideas of detecting different types of memory errors.

A. *PMU-based Sampling*

`Sampler` proposes to leverage the Performance Monitoring Unit (PMU) hardware to sample memory accesses in order to detect buffer overflows and use-after-frees.

The PMU hardware can sample memory accesses or other hardware-related activities [4], [28]–[34]. Currently, the PMU hardware is ubiquitous in modern architectures, including Instruction-Based Sampling (IBS) [35] in AMD Opteron processors, and Precise Event-Based Sampling (PEBS) [36] in Intel-based processors. The PMU hardware will sample memory loads and stores, with their precise instruction pointers (and call stack), timestamps, and memory addresses. The memory address can be utilized to determine an invalid access, while the precise instruction pointer (IP) can be used to infer the line-of-code information of an application. The timestamp information reports the temporality of each event, which could be utilized to confirm whether an access is a use-after-free.

Operating the PMU hardware requires kernel support. Fortunately, the Linux kernel has provided such support since 2009 (Linux-2.6.31) [37]. Users could program the corresponding registers using the `perf_event_open` system call. After obtaining a file descriptor returned from this system call, a communication buffer can be established between the user space and kernel space using the `mmap` system call. Based on the preset sampling period, the kernel can write each sampled event into this shared buffer, then interrupt the user program to collect the sampled events when the buffer is full. For instance, if the sampling period is set to 5000, the PMU hardware will sample one out of 5000 memory accesses. The overhead and effectiveness of PMU-based sampling highly depends on the sampling period: more frequent sampling may introduce a higher overhead and a better understanding of the execution, while less frequent sampling reduces both overhead and effectiveness.

B. *Basic Ideas Of Detection*

`Sampler` is designed to detect a range of memory errors, including buffer overflows, use-after-frees, double and invalid frees, where the basic ideas are described as follows.

1) Detecting Buffer Overflows

Buffer overflows occur when programs read/write outside the boundaries of an object or variable. Buffer overflow is a well-known source of security attacks [1] and many reliability issues [2].

To detect buffer overflows, existing work such as AddressSanitizer and DoubleTake allocate additional redzones (or canaries) around each heap object [2], [16], as shown in Fig. 1. If the redzone is touched, the buffer overflow is detected.

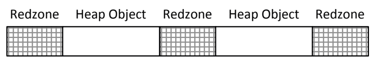

Fig. 1. Adding redzones along with objects.

`Sampler` borrows the redzone idea: if a sampled access is found to read or write within the redzone, an overflow is detected and reported. `Sampler` organizes heap objects by power-of-two sizes, and adjusts the size of each allocation if necessary in order to make room for the redzone: if an application requests a size less than the nearest power-of-two ceiling, all remaining bytes of this object will be treated as the redzone; otherwise, the allocation will be satisfied from the next power-of-two size class, leaving the same space as the requested size for the redzone. Additionally, `Sampler` utilizes a different method to mark redzones, as described in Section II-C, instead of using the bitmap mechanism.

2) Detecting Use-After-Frees

Use-after-frees (or dangling pointers) occur whenever an application accesses a previously-deallocated object, while this object is currently utilized for other purposes. Use-after-frees may lead to unexpected program behavior or be exploited to breach security.

In order to detect use-after-frees, `Sampler` borrows the idea of AddressSanitizer: freed objects will be treated as redzones, where touching them will be reported as a use-after-free. To further increase the capability of detection, freed objects are placed in a "quarantine list" so that they cannot be reutilized in the near future. These objects are actually freed (and thereafter available to be re-utilized) when the total size of all quarantined objects rises above a predefined threshold, or when there are no available slots in the given quarantine list. However, `Sampler` utilizes per-thread quarantine lists in order to reduce contention caused by placing objects into the same list. As with the detection of buffer overflows, `Sampler` also employs a different mechanism to mark redzones, described in Section II-C.

3) Detecting Double and Invalid frees

A double-free problem indicates that an object is freed more than once, while an invalid-free is caused by passing an invalid pointer to the `free` function. Both errors have been exploited to perform security attacks [38].

`Sampler` is guaranteed to detect all double and invalid frees accurately, with no false positives. It employs its custom allocator (described in Section III) for detection, without requiring use of the sampling mechanism. Basically, `Sampler` embeds the status information of each object into its metadata. A double-free is detected if a pointer passed to the `free` function is currently referring to a freed object, where the mechanism of delaying re-allocations helps to catch such bugs. To detect invalid frees, `Sampler` first checks whether the address belongs to the heap, then checks whether the object is a valid heap object. Any violation will indicate an invalid-free problem. `Sampler`'s allocator provides the information-computable capability, which allows for the fast lookup of metadata and confirmation as to whether an address corresponds to a valid heap object.

In summary, `Sampler` applies the redzone idea to sampling-based memory error detection. `Sampler`'s novel custom allocator enables it to perform faster checking and avoid false alarms.

C. Technical Challenges

As described in Section I, there are multiple technical challenges, as listed below, along with the corresponding sections in which they are addressed.

- How to avoid the corruption of heap metadata, and therefore prevent false alarms caused by this? `Sampler` designs a custom memory allocator that separates the metadata from the actual heap, as described further in Section III.
- How to reduce or prevent the overhead caused by the use of bitmaps? For instance, AddressSanitizer adds at least $1/8$ of its memory usage due to the bitmap. `Sampler` relies on its information-computable capability, and places the last valid address of each object into the metadata. `Sampler` checks buffer overflows and use-after-frees by comparing each memory reference with this address: any references beyond the starting address and the last valid address will be considered an invalid operation and should be reported.
- The checking of bitmaps may incur significant performance overhead. Again, this issue is addressed by the information-computable capability. `Sampler` supports fast metadata lookup, as described in Section III.
- How to crowdsource the detection of different executions? `Sampler` starts sampling randomly for different threads and different executions, in order to cover as many different sequences of memory accesses as possible, which helps to detect more memory errors. This is further described in Section IV-B.
- How to reduce the overhead caused by frequent interrupts? `Sampler` cumulatively collects and analyzes samples as described in Section IV-C.

III. Custom Memory Allocator

`Sampler` designs a novel memory allocator to overcome the issues associated with employing the Linux allocator to meet its unique needs for dynamic memory management. `Sampler` takes separate approaches to manage small and large objects, similar to existing allocators [39], [40]. An allocation request larger than one megabyte will be satisfied in the large heap (as described in Section III-B). Otherwise, it will be treated as a small object, and managed using the scheme described in Section III-A.

A. Management of Small Objects

For small objects, the memory layout of the allocator is illustrated as Fig. 2. The allocator takes advantage of the vast address space of 64-bit machines, and obtains a large block of memory from the underlying operating system initially. This large block is further partitioned into multiple chunks, where each chunk is called a "bag" throughout the remainder of this paper. Currently, the size of each bag is 32 gigabytes, but is an adjustable option at compile time.

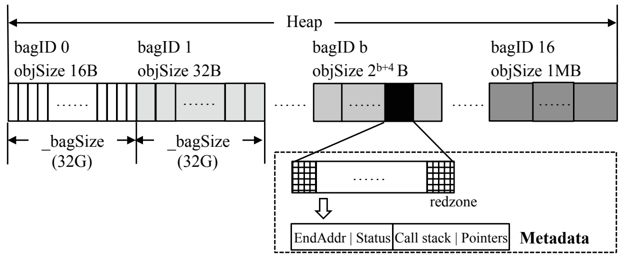

Fig. 2. `Sampler`'s custom allocator for small objects.

`Sampler` also employs the idea of BIg-Bag-Of-Pages (known as "BIBOP" style) to manage these bags [38], [41]. Basically, all pages within the same bag hold objects of the same size. The size of these heap objects are always a power-of-two, ranging from 16 bytes up to one megabyte. Note, that the object size of these bags is monotonically increasing from 16 bytes to one megabyte. That is, the first bag will be dedicated for objects of 16 bytes, the second bag will hold objects of 32 bytes, and so forth. This design will not allow for two bags holding objects of the same size. Also, these objects are never further divided or coalesced, which is different from the Linux allocator. If an allocation request for a specific size cannot be satisfied, `Sampler` should report this to the user. Then, the size of each bag should be changed to a larger value upon restarting this application. However, 32 gigabytes for each size class should be sufficiently large for most applications, as we have never encountered a situation exceeding this.

Since each bag is always holding objects with the same size, it is possible to separate the metadata of all objects from the actual heap. This design will prevent metadata corruption caused by memory errors, such as buffer overflows. Also, this layout actually provides the **information-computable capability**: given an address within the heap, we can easily compute the starting address, size class, and the metadata placement of the corresponding object, if we know the starting addresses of the heap and the metadata.

The allocator employs a per-thread sub-heap design, such that memory allocations by each thread will be satisfied from their own sub-heap [39]. This method reduces the false sharing effect that may occur when multiple threads operate on the same cache line simultaneously [42]. In `Sampler`'s design, there is no need to acquire locks upon every allocation and deallocation, except in cases of memory blowup as described below: each allocation request will be satisfied from each thread's per-thread heap, either from its bump pointer or its free list; conversely, each deallocation request will be inserted into each thread's free list that corresponds to the object's size class.

a) Allocation

Each thread's sub-heap has two mechanisms to satisfy an allocation request: one is to use the bump pointer that always points to the next never-allocated object on the heap, while the other is to use the free list that tracks free objects from the current thread [43]. Upon each allocation, the free list will be checked first. Typically, objects in the free list are managed in Last-In-First-Out (LIFO) order, where the most recently deallocated object will be re-utilized first. This method will benefit temporal locality, since the recently-deallocated object is typically still in the cache. If there are no objects available in the corresponding free list, we will allocate the object pointed to by the bump pointer, then increment the bump pointer to refer to the next object. If the sub-heap is exhausted, i.e. there are no available objects in the free list and the bump pointer is invalid, another block (e.g. 8MB) will be obtained from the corresponding bag, under the protection of a bag-wise lock.

b) Deallocation

Each deallocation will be returned to the current thread's free list related to a particular size class. This method avoids unnecessary lock acquisitions and possible lock contention. However, it will not generate unnecessary cache contention: when an object is freed by another thread, this freeing thread has typically loaded the cache line already, which will not cause unnecessary cache contention even if the object is re-used by the thread. Furthermore, it will not exacerbate the false sharing problem, in comparison to the approach of returning a freed object to its owner thread: two different threads may work on the same cache line for both cases. We have experimentally confirmed that this method improves performance.

c) Information-Computable Capability for Fast Error Checking

`Sampler`'s allocator accelerates the metadata lookup by computing related information (such as bag ID, object size, and object ID) directly from the sampled memory address.

Fig. 3 (heap layout) and Fig. 4 (pseudo-code) show how `Sampler` computes them. For each sampled address ($sampleAddr$), the bag index ($bagID$) to which the sampled access belongs can be computed by dividing the offset within the heap ($sampleOffset$) with the fixed-length bag size (_$bagsSize$, 32GB). Because bags are allotted together with a monotonically increase, we can compute the object size ($objSize$) using the bag index ($bagID$). Since all objects in a bag have the same size, by using the offset within the bag ($sampleOffset$ - $bagOffset$), we can determine the object index ($objID$) in turn. It is also possible to compute the object starting address ($objStart$) using the precomputed values ($bagOffset$, $objID$, and $objSize$). Most importantly, this fast-calculated unique $objID$ allows `Sampler` to directly look up the metadata with no searching, enabling fast memory error checking.

Although `Sampler`'s allocator shares the same "information-computable capability" as FreeGuard [44],

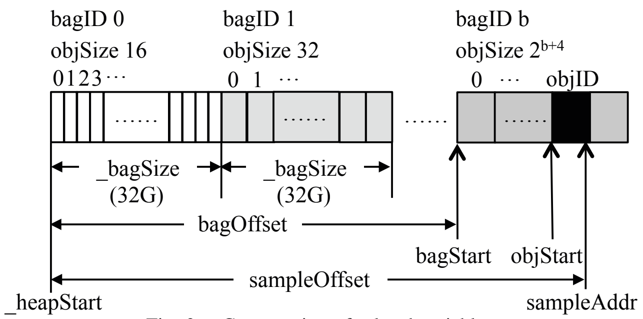

Fig. 3. Computation of related variables.

```
sampleOffset  = sampleAddr - _heapStart;
bagID         = sampleOffset / _bagSize;
bagOffset     = bagID * _bagSize;
objSize       = 1 << (bagID + 4);
objID         = (sampleOffset - bagOffset) / objSize;
objStart      = bagOffset + objID * objSize
              + _heapBegin;
objMetadata   = _bagMetadata[objID];
```

Fig. 4. Computing information about `sampleAddr`.

it is distinct in three aspects: (1) FreeGuard will always return the freed object back to its owner, which may introduce unnecessary overhead caused by lock acquisitions and lock contention; (2) `Sampler` can compute the information faster: given an address within the heap, it requires only six instructions to determine the size class and starting address of each object, which is much faster than FreeGuard; (3) In FreeGuard, it is possible to determine the owner thread's information for each object, while `Sampler` cannot do this, since it does not have to. Overall, `Sampler`'s allocator is simpler than FreeGuard, and also reduces the amount of computation steps by almost half.

d) Reducing Memory Blowup

Memory blowup – where memory consumption is unnecessarily increased because freed memory cannot be reused to satisfy future memory requests [39] – may occur, since freed objects are always returned to the current thread's sub-heap, especially for applications using the producer-consumer model. Therefore, a donation mechanism is further designed to reduce the memory blowup. Each thread will monitor the total size of freed objects for each size class, and start to donate partial objects into the global free list, if the total size of freed objects is larger than a predefined threshold, such as 64 KB. Different from existing work [45], this threshold is adaptively changed: if the amount of deallocations is less than twice of allocations in this thread, we will double the threshold, making it less likely to donate. Typically, a thread will not donate freed objects, except for when the producer-consumer model is used. We have confirmed that this mechanism will significantly reduce unnecessary donations, avoiding unnecessary migration of objects across multiple threads.

B. Management of Large Objects

`Sampler` manages large objects (those with sizes larger than 1 MB) in another separate heap. The number of large objects is expected to be far fewer than that of small objects. Therefore, we do not use the per-thread heap design to avoid lock contention here. Instead, all threads will share the same heap, avoiding the performance loss caused by unnecessary cache loading and page faults. Large objects are aligned to 1MB, instead of using the power-of-two size classes as with small objects.

In order to further reduce cache loadings, each thread will still prefer to re-utilize objects freed by itself. Each thread maintains one free list, which tracks the objects deallocated by the current thread. Upon allocation, each thread will first allocate from its free list, if possible. Whenever an available object is found in its free list, it will check the status of its neighbors in order to find a block with sufficient size. If this is the case, then multiple contiguous blocks will be coalesced together to satisfy the request. However, if the current free list is empty, or if there are no available objects that satisfy the request, even with coalescing, then `Sampler` proceeds to search, beginning from the starting address of the big heap. Only when these mechanisms fail, `Sampler` will use the bump pointer to allocate from the never-allocated area. Each 1MB block of memory will maintain its corresponding metadata, which indicates the last valid address inside this block, as well as status information, which will be utilized during error checking. For large objects, the status information of the object should be propagated to every coalesced 1MB block upon allocations and deallocations.

IV. Implementation Details

This section describes how `Sampler` works to detect different types of memory errors, based on the combination of PMU-based sampling and custom allocator design.

A. *Intercepting Allocations and Deallocations*

`Sampler` intercepts memory allocations and deallocations using the preloading mechanism. `Sampler` places redzones upon allocations for detecting buffer overflows and use-after-frees, while it detects double and invalid frees upon deallocations (see Section II-B3).

As described in Section II-B, redzones are placed after each contiguous object to detect both buffer underflows and overflows. `Sampler` uses flexible sizing of redzones by interacting with its custom allocator. It always aligns the size of each allocation to the next-highest power-of-two, and dedicates all remainder bytes beyond the requested size of the object as redzones. If an allocation is already aligned to a power-of-two size, the object will be allocated from the next-largest size class, which will install a memory region equal to the requested size as redzones. On deallocation, every freed object is moved into a quarantine list and marked as redzones (i.e., the entire object is rendered inaccessible) in order to detect use-after-frees.

In particular, to track redzones, `Sampler` utilizes a different mechanism from [2], [16] that use the bitmap mechanism. When using a bitmap, the bits representing redzones must be marked explicitly upon (de)allocations, imposing unnecessary memory and performance overhead. However, existing work

```
ldFd=perf_event_open(&aLd,0,-1,-1,0);
stFd=perf_event_open(&aSt,0,-1,ldFd,0);

ringBuf=mmap(NULL,MAPSIZE,PROT_READ
       |PROT_WRITE,MAP_SHARED,ldFd,0);

auxBuf=mmap(NULL,2*AUXSIZE,PROT_READ
   |PROT_WRITE,MAP_SHARED,ldFd,MAPSIZE);

fcntl(ldFd,F_SETFL,O_NONBLOCK|O_ASYNC);

ioctl(stFd,PERF_EVENT_IOC_SET_OUTPUT,ldFd);

fcntl(ldFd,F_SETOWN_EX,&owner);
fcntl(stFd,F_SETOWN_EX,&owner);
fcntl(ldFd,F_SETSIG,SIGIO);
fcntl(stFd,F_SETSIG,SIGIO);
```

Fig. 5. Initialization of the PMU driver for each thread.

cannot avoid using a bitmap, since objects of different sizes are physically placed together. For each sampled memory address, it is thus impossible to know the starting address of the object accessed by the sample. On the other hand, `Sampler` does not maintain a bitmap. As `Sampler` can compute the size and starting address of the object (Fig. 4), it places the last valid address of the object into its metadata (Fig. 2). This speeds up the checking procedure and obviates updating the bitmap upon (de)allocations; any access beyond this last valid address will be considered a buffer overflow. For detecting use-after-frees, the last valid address of the current object will be set to one less than the current address, which treats any memory reference on the object as a use-after-free.

B. *Sampling Memory Accesses*

During program execution, the PMU hardware samples the user-level retired load and store events, and these sampled events are then analyzed to identify buffer overflows and use-after-frees. The PMU configuration of the sampling mechanism is further described as follows.

`Sampler` randomly starts sampling for different threads and during different executions, but using the same sampling period, to avoid the same sequence of the sampled memory accesses. `Sampler`'s method helps capture different sequences of memory accesses within different threads and different executions, helping to detect as many errors as possible.

To initialize the PMU driver, `Sampler` utilizes the `perf_event_open()` system call to set up the PMU hardware, as illustrated in Fig. 5. Currently, `Sampler` is evaluated on Intel's Skylake architecture, where the event type of loads is set to $0x108081d0$, and the one for stores is set to $0x82d0$. There are multiple notes about this. First, every thread should perform such initialization. The flags passed to `perf_event_open()` guarantee to measure the events of the calling thread on any core, avoiding possible effects caused by thread migration. Second, to avoid unnecessary interference with other threads, different threads are set to handle the I/O signal (`SIGIO`) by themselves. This is achieved by invoking the `fcntl` system call with the `F_SETOWN_EX` flag. Third, load and store events should be initialized separately by invoking `perf_event_open()` and `fcntl`. Both type of events will be dumped into the same buffer, which simplifies their handling; however, these events can still be differentiated by checking the "status" field of each sample's PEBS record. The file descriptor for load events is passed to `perf_event_open` when initializing store events, and `ioctl` is invoked with `PERF_EVENT_IOC_SET_OUTPUT` to connect these two types of events. Fourth, `mmap` system calls generate a ring buffer and an auxiliary buffer that will be shared by the kernel and user applications, such that samples may be obtained by reading through this shared buffer.

During the initialization, the "precise_ip" attribute should be set to the maximum value, since the instruction pointer skid controls the number of instructions that occur between the eventing IP and the recording of the event [46]. The CPU attaches the timestamp to the raw sample using the value of the TSC counter, in order to determine the temporal relationship between each reference and corresponding free operations.

Reducing Sampling-related Overhead

`Sampler` takes multiple approaches to further reduce the sampling overhead.

First, it borrows the idea of ProRace to reduce unnecessary format translations and kernel-to-user copies [47]. Currently, the PMU hardware automatically stores samples to a kernel-space buffer, known as the Debug Store (DS), typically with a size of 64KB under Linux. When the DS is nearly full, an interrupt will be raised so that the kernel may add additional information to each sampled event, such as wall-clock time and sample size, generating samples in the format of "`perf` events". Then, these events will be copied to the user/kernel shared area, which allows the user-land `perf` tool to analyze them. However, this procedure involves unnecessary formatting to translate raw PEBS events into "`perf`" events, and an unnecessary copy from the DS to the ring buffer. The ProRace PEBS driver avoids this overhead by only storing raw PEBS events, and designs an aux-buffer that eliminates the additional copy from the DS area to the shared ring buffer. Finally, the kernel sends a `SIGIO` signal with the `POLL_HUP` flag, informing the user application that it can obtain these samples by reading from the shared buffer. Their aux-buffer includes two 32KB segments for PEBS events, which will be swapped when the current segment is full. However, this mechanism increases the complexity unnecessarily, and can also be unstable. `Sampler` modifies the mechanism to use only a single segment, but with a larger size (64KB).

Second, `Sampler` only analyzes the sampled events online, without the need for saving events to disk. Typically, writing events to disk may incur prohibitive performance overhead caused by file I/O operations. As described above, this online analysis also improves convenience, due to avoiding unnecessary offline analysis. Furthermore, from a security standpoint, such an online analysis is rather essential to enable timely attack detection.

Third, `Sampler` reduces the number of interrupts. The kernel will only notify the user applications when the aux-buffer is nearly full. Delaying the checking of memory references may present some challenges to guaranteeing the correctness of reports, as further described in Section IV-C.

C. Collecting Memory Samples

To avoid significant performance overhead caused by frequent interrupts, `Sampler` accumulatively collects memory samples in three cases: (1) when the aux-buffer is full, the kernel notifies it using the `SIGIO` signal. (2) `Sampler` polls the samples when the quarantine list reaches capacity. (3) `Sampler` collects samples upon normal and abnormal exits of applications.

Case (1): The signal handler will read all samples in the aux-buffer when the buffer is full. For each sampled memory reference within the range of the heap, the handler will check the address, the timestamp, and the access type (load or store) to confirm whether the current reference is a buffer overflow or use-after-free error. The detection of memory errors is further described in Section IV-D. Afterward, `ioctl` is invoked with `PERF_EVENT_IOC_REFRESH` in order to resume sampling once again.

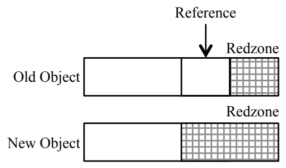

Fig. 6. Potential False Positive with Naive Object Reutilization.

Case (2): The polling method is only invoked when the quarantine list of any thread becomes full, where this thread becomes the coordinator thread. As described in Section II-B2, quarantine lists hold freed objects so that these objects are not re-utilized too quickly. Then, any reference on these freed objects (marked as redzones) are use-after-free errors. When one quarantine list is full, freed objects will be actually freed to make room for newly-freed objects. Here, the coordinator thread notifies other threads to stop their executions and poll existing samples simultaneously. Otherwise, false alarms can be generated, as shown in Fig. 6. When a freed object is re-utilized (e.g., allocated) as the "New Object", a valid reference on the "Old Object" will be reported as a buffer overflow, provided that the sampled reference is processed after the reallocation. To avoid any issues related to this, `Sampler` further introduces a barrier to guarantee that all threads will proceed to normal executions only after every thread has finished reading all samples.

Case (3): `Sampler` also analyzes samples when a program quits, including normal and abnormal exits. `Sampler` registers a function that will be called on normal exits, by invoking the `atexit` API explicitly before entering the `main` routine. `Sampler` also registers the signal handlers for abnormal exits, such as the `SIGSEGV` signal. When exits are caused by some program errors, `Sampler` prints the call stack of the faults. For both cases, `Sampler` performs error detection by reading samples using the polling method.

D. Detecting Memory Errors

For any sampled memory access, `Sampler` should determine whether an access is a buffer overflow or use-after-free error, or just a normal access. Due to a large volume of sampled accesses – thousands per second – the error checking should be done very efficiently. `Sampler` uses two mechanisms to accelerate the checking procedure: (1) its custom allocator provides the "information-computable" capability, such that the starting address of each object and its metadata information can be computed quickly. Therefore, there is no need to check an additional bitmap for the placement of redzones, avoiding the loading of additional cache lines. (2) `Sampler` places the last valid address of the current object into the metadata, and the checking takes only one comparison to determine validity: a reference on an address less than the last valid address is considered to be valid. There is no need for further confirmation if the reference is valid, which is true for most references. Otherwise, we need to check errors further.

If the current object is not freed, then the invalid reference is a buffer overflow. `Sampler` guarantees not to free the object until all existing samples have been analyzed. Therefore, it never generates false positives, as illustrated in Fig. 6. If the object has been freed, then we should further confirm whether it is a use-after-free error. `Sampler` compares the timestamp of the reference and the corresponding free operation to determine this. The PMU hardware attaches the TSC timestamp for each event, and `Sampler` always records the timestamp of free operations into the metadata of each freed object in the quarantine list. Therefore, it is possible to compare these two timestamps to determine whether the reference occured after the free operation (use-after-free).

However, there are multiple potential issues here. First, older versions of hardware (e.g. Ivy Bridge) cannot attach the timestamps to each event, which is performed when the kernel is interrupted to transform the events. This will generate false positives, since the reference may contain a later timestamp than it should. Fortunately, newer versions of hardware (at least after Skylake) will attach the TSC timestamp for each sampled event. This allows us to utilize the timestamp to reliably identify use-after-frees correctly. Second, different cores may have different TSC timestamps at the same time. However, the difference between a pair of cores are invariant with modern hardware support, which is known as "invariant TSC" [48]. Therefore, `Sampler` evaluates the difference beforehand, and the utilizes a reasonable value as the threshold (around 1000). If a reference occurs within the threshold, it is not considered a use-after-free. Conceptually, it is possible for `Sampler` to miss the immediate uses after a free operation. However, this will happen very rarely, as it requires the free and use operations from different threads to occur very closely in time. Use-after-free within the same thread does not suffer from the same issue. Furthermore, there are generally multiple accesses for use-after-free errors.

E. *Reporting Memory Errors*

Currently, `Sampler` simply reports these errors on the screen. In the future, this report can be sent to the developers via email, after obtaining user approval.

For use-after-free problems, `Sampler` reports the call stack of memory allocation and deallocation, and the precise instruction causing the error. For buffer-overflows, `Sampler` reports the call stack of the corresponding memory allocation, as well as the statement causing the actual overflow. Similarly, `Sampler` reports the call stack when detecting double and invalid frees.

F. *Supporting Multithreaded Applications*

To support multithreaded programs, `Sampler` intercepts the `pthread_create` function in order to assign the heap for each thread, initialize per-thread data structures, set up the sampling mechanism, and utilize a custom function for the task of thread creation. The custom function actually invokes the real thread function, which also allows `Sampler` to capture the exit of threads. Inside `Sampler`, an internal thread index is utilized to indicate each thread.

Fetching per-thread data quickly: During every memory (de)allocation, `Sampler` will obtain the index of the current thread to allocate from and return objects to its per-thread heap. Upon each interrupt, `Sampler` should also use this index to determine the placement of a thread's corresponding ring buffer. Thus, there is a large number of times that we must acquire the internal thread index. We could naively rely on the thread-local storage area, which can be declared using the keyword "`__thread`" [49]. However, this naive method may involve at least the cost of an external library call and a lookup in the indexed table. Instead, `Sampler` borrows the method of existing work to circumvent the cost of using TLS variables [50]. `Sampler` assigns the stack area for each newly-created thread upon thread creation. The stack area of all threads (except the main thread) will be allotted contiguously, and the offset of each thread's stack starting point will be equal to the product of its thread index and stack size. Thus, `Sampler`can compute the thread index by dividing the offset between a stack variable and the starting address of the global stack map with the stack size.

V. Discussions

A. *Detection Effectiveness*

`Sampler` proposes a sampling-based mechanism to detect heap overflows and use-after-frees, while it is guaranteed to detect all double and invalid frees due to the design of its allocator.

For heap overflows and use-after-frees, the detection effectiveness is proportional to the sampling rate. For instance, if the sampling period is 5000, then `Sampler` has a sampling rate of 0.02%. Therefore, `Sampler` is able to detect heap overflows and use-after-frees with a probability of at least 0.02%. In reality, both buffer overflows and use-after-frees may incur multiple references, which explains why `Sampler` has a much higher probability of detecting these errors (as evaluated in Section VI-B).

B. *Limitations*

For the time being, `Sampler` has the following limitations. First, `Sampler` does not handle buffer overflows on stack and global variables. To support these, `Sampler` should instrument the code in order to place redzones between different variables, as performed by AddressSanitizer [2].

Second, `Sampler` cannot detect all occurrences of memory errors on each execution due to its sampling nature. `Sampler` compromises its detection effectiveness by trading for lower overhead, in order to be capable of deployment in production software. However, `Sampler` is able to detect all bugs, given a reasonable number of executions.

VI. Evaluation

This section aims to answer the following questions:

- **Effectiveness:** Can `Sampler` detect memory errors in real-world applications?
- **Performance Overhead:** What is the performance overhead for `Sampler`, and how does `Sampler`'s allocator perform separately?
- **Memory Overhead:** What is the memory overhead of `Sampler`? How much is attributed to its allocator?

A. *Experimental Setup*

We performed experiments on a 4-core quiescent machine, with an Intel® Core™ i7-6700K CPU processor running at 4.00GHz, which is a Skylake model. This machine has 16GB of main memory, and 64KB L1, 256KB L2 and 8MB L3 cache, separately. The experiments were performed on Linux-4.5.0, with a patch to include the Pro-Race PEBS driver. We used GCC-5.4.0 with `-O2`, `-g` and `-fno-omit-frame-pointer` flags to compile all applications. AddressSanitizer was compiled using Clang 6.0.0, and built with detection for only heap-based overflows and use-after-free bugs, in order to provide a fair comparison.

B. *Effectiveness*

We performed the effectiveness evaluation on 12 real bugs in `PHP 5.6.3`, `bc`, `gzip`, `polymorph`, `libtiff`, `bzip2`, and `ed`. The PHP interpreter is supplied with different bug-triggering input files, which will exercise the code with different use-after-free errors. In total, we evaluated 12 bugs, including one double-free, one invalid-free, five use-after-frees, and five buffer overflows. Due to the fact that different sampling periods could significantly affect the effectiveness, we have evaluated three different settings: p=1000, p=5000, and p=10000. p indicates the sampling period, which is inversely proportional to the sampling rate. For instance, when the sampling period is 1000, the PMU hardware will sample one access out of every 1000 instructions. Therefore, "p=1000" actually has the highest sampling rate (0.1%), while "p=10000" has a sampling rate of 0.01%.

TABLE II shows the number of times that the specified bug is detected out of 1000 executions. We observe multiple facts from these results: (1) `Sampler` can always detect double and invalid frees, regardless of sampling period. This is due to the special design of `Sampler`'s custom allocator. (2) A higher sampling rate typically implies a higher effectiveness of detecting use-after-frees and buffer overflows. (3) Typically, the detection rate is much larger than the sampling rate. In general, the sampling period of 5000 presents both reasonable overhead (as evaluated in Section VI-C) and reasonable effectiveness, which is the default sampling period. There are no false negatives when combining 1000 executions together, when using this sampling period. The sampling period is a macro that users may change easily at compile time. With this sampling period, `Sampler` detects use-after-frees in a range of between 1.9% and 3.3%, with an average of 2.6%. For buffer overflows, `Sampler` has a detection rate between 0.08% and 99.5%, with an average of 48.1%. If the user would like to pay more performance overhead (less than 7%), `Sampler` could use the sampling period of 1000, which will boost the average detection rate to 6.3% and 80.5% for use-after-frees and buffer overflows respectively. We also confirm the reason for a high detection rate, often much higher than expected for the sampling rate, is due to the fact that most bugs touch redzones multiple times.

The column "First Detection" of TABLE II shows when the corresponding bug will be first detected, with the default sampling period (5000). For use-after-free bugs, it takes around 46 executions to detect the bugs inside, while around 98 executions are required to detect buffer overflows. Note that `Sampler` may utilize different numbers of executions to detect the bug, due to its randomization mechanism.

Probability Analysis:

We further formulate the detection probability as follows:

The bug detection can be represented by a Bernoulli trial with two possible outcomes, "success" or "failure", where the probability of successful detection is fixed and each execution is independent. Given a sampling period P and the number of invalid accesses i that occur during an execution, the probability of a successful detection equals:

$$p = 1 - (\frac{P-1}{P})^i$$

Therefore, we can predict the number of Bernoulli trials X expected to obtain the first success:

$$Pr(X = k) = (1-p)^{k-1}p = (\frac{P-1}{P})^{i(k-1)}(1 - (\frac{P-1}{P})^i)$$

Further, the probability of obtaining at least k detections given n trials can be computed using the following equation:

$$Pr(X \geq k; n, p) = 1 - \sum_{j=0}^{\lfloor k \rfloor} \binom{n}{j} p^j (1-p)^{n-j}$$

In the end, the probability of obtaining a false negative result, given a number of executions n, each with a probability of success p, is equal to:

$$Pr(X = 0; n, p) = \binom{n}{0} p^0 (1-p)^{n-0} = (1-p)^n$$

We could examine the `bc` bug as an example, which has $i = 32$ invalid accesses. Given the parameters $P = 5000$, the expected first success is around $\approx$ 157, while the observed value was 292. The expected number of detections out of 1000 executions is $np = 6.3802$, while the actual observed value for the number of detections was 8.

The `gzip` bug will have 3074 invalid accesses in total. Therefore, the probability of observing at least 447 successes is approximately equal to 0.7794, given $n = 1000$ and $P = 5000$. Further, the expected number of detections out of 1000 executions is 459.2842, while the observed number of successful trials was 447. Finally, the predicted first detection is 2.1773, while the observed value was 1. This also indicates that `Sampler`'s sampling rate is not throttled by the underlying OS, which enables reliable performance for a particular sampling rate.

If we examine the case of a bug such as `bzip2`, with a number of invalid accesses of $i = 48$ and a sampling period of $P = 5000$, we determine the probability of obtaining a false negative result for $n = 500$ executions to equal 0.008226. Similarly, for $n = 1000$ and $n = 2000$, these probabilities are 0.000068 and 4.5784×10^{-9}, respectively.

Therefore, these results indicate that `Sampler` is able to detect all memory errors when combining a sufficient number of executions together.

C. *Performance Overhead*

We evaluated the performance of `Sampler` on the PARSEC [52] multithreaded benchmark suite, as well as real applications such as `Aget 0.4.1`, `Memcached 1.4.25`, `MySQL 5.6.10`, `Pbzip2 1.1.6`, `Pfscan 1.0`, and `SQLite 3.12.0`.

Fig. 7 shows the performance overhead of ASan, `Sampler`, and Perf, where all values in the figure represent the average of 10 executions. Both `Sampler` and Perf use the same sampling period – 5000. The figure shows "Normalized Runtime" which represents the runtime normalized to that of the default Linux libraries. Therefore, a lower bar indicates better performance. For these systems, ASan is the state-of-the-art in detecting memory errors [2], which instruments every memory access for its detection. `perf` is a utility that simply collects memory references with the PMU mechanism, but without detecting errors. Naively, `perf` could be employed to build the detection tool.

Comparing to the default Linux libraries, `Sampler` imposes around 2.4% performance overhead on average, when utilizing the default sampling period of 5000. In contrast, ASan runs about 45% slower on average, whose high overhead indicates its inapplicable usage for deployment, and the motivation for `Sampler`. `perf` imposes over 2× performance overhead, which indicates that the naive mechanism of employing the PMU for detection is not sufficient. `Sampler`'s custom memory allocator, its mechanisms for collecting sample data, and its mechanism of reducing translation and kernel-to-user copies (borrowed from ProRace) work together to reduce the performance overhead.

`Sampler` only imposes overhead larger than 10% for one application. This application is `canneal`, which has a large number of memory allocations, exceeding 30 million in total. Therefore, obtaining the call stacks of these allocations and deallocations can be a significant source for increased overhead. The high overhead of `raytrace` and `swaptions` can similarly be attributed to this reason. We also observe that `Sampler` actually achieves some performance speedup for `bodytrack`, and near zero overhead for others, such as `facesim` and `streamcluster`. These speedups come from `Sampler`'s custom memory allocator, which can be seen in the "SA Allocator" series of Fig. 7. To evaluate `Sampler`'s allocator, we exclude all detection logic; that is, no quarantine list, no sampling, and no collection of call stacks for allocations and deallocations. In fact, `Sampler`'s allocator is actually running 2.7% faster than the Linux allocator. Based on our understanding, multiple factors contribute to the excellent performance of `Sampler`'s allocator: it generally avoids the use of locks for most allocations and deallocations, and it separates metadata from the actual heap, which improves cache utilization since the same quantity of cache will hold more frequently-accessed data, rather than metadata, whose accesses are less frequent.

Impact of Sampling Rate: Similar to the effectiveness evaluation, we evaluated the performance overhead of `Sampler` using three different settings: p=1000, p=5000, and p=10000. The performance results are shown in Fig. 8. For a sampling period of 1000, the average overhead is around 6.6%, while the overhead is 1% with a sampling period of 10000. Therefore, users may determine their sampling rate by their performance budget, since typically more frequent sampling indicates a higher overhead, but with the better effectiveness (as shown in TABLE II).

D. *Memory Overhead*

The memory overhead of `Sampler` was evaluated using the same applications as the performance evaluation, with the default sampling period—5000. To our understanding, different sampling rates will not significantly affect the memory overhead, which is why we only include an evaluation for this sampling period. Memory consumption is collected via the Linux `time` utility. For server applications that do not terminate, we implement a script to periodically obtain the `/proc/`*`PID`*`/status` files, then display the maximum resident set size (RSS) of the process throughout its lifetime.

The results of memory overhead are shown in TABLE III. In total, `Sampler` imposes around 76% memory overhead, while `Sampler`'s allocator imposes 34% memory overhead, when comparing to the default Linux allocator. The state-of-the-art, ASan, imposes over 2× memory overhead for the same applications. This indicates `Sampler`'s memory overhead is still acceptable compared to the state-of-the-art. We also noticed that `Sampler` adds a high percentage of startup memory overhead for small-footprint applications (e.g. `swaptions`). However, for applications with a larger footprint, the memory overhead is typically reasonable, and adjustable using the quarantine list configuration parameters.

Comparing to the Linux allocator, the `Sampler` allocator utilizes additional memory to store free list pointers, and always allocates objects in power-of-two sizes, which adds some alignment overhead. To this, `Sampler` adds around 42% memory overhead. Both the per-thread quarantine lists and allocator metadata will contribute to `Sampler`'s memory overhead. Because the size of the quarantine list is customizable, the actual overhead may vary depending on different

			# Detections			
Application	Type	Bug	P=1000	P=5000	P=10000	First Detection
PHP 5.6.3	Double free	CVE-2016-5772	1000	1000	1000	1
PHP 5.6.3	Use-after-free	CVE-2016-6290	45	19	1	19
PHP 5.6.3	Use-after-free	CVE-2016-5771	94	26	1	96
PHP 5.6.3	Use-after-free	CVE-2016-3141	64	27	1	59
PHP 5.6.3	Use-after-free	CVE-2015-6835	57	33	8	29
PHP 5.6.3	Use-after-free	CVE-2015-0273	56	26	6	37
bc 1.06	Buffer overflow	Bugbench [51]	26	8	0	292
bzip2 1.0.3	Buffer overflow	Bugbench [51]	999	853	861	1
gzip 1.2.4	Buffer overflow	Bugbench [51]	1000	447	124	1
libtiff 4.0.1	Buffer overflow	CVE-2013-4243	999	995	977	1
polymorph 0.4.0	Buffer overflow	Bugbench [51]	1000	105	2	193
ed 1.14.2	Invalid free	CVE-2017-5357	1000	1000	1000	1
AVERAGE			528.33	378.25	331.75	60.83

TABLE II
DETECTION EFFECTIVENESS RESULTS FOR `SAMPLER` USING 1000 EXECUTIONS. `SAMPLER` NEVER REPORTS FALSE POSITIVES USING THE NEWEST HARDWARE SUPPORT AND WITH THE ASSISTANCE OF ITS CUSTOM ALLOCATOR. "FIRST DETECTION" INDICATES THE EXECUTION IN WHICH SAMPLER (USING $P = 5000$) FIRST DETECTS THE BUG.

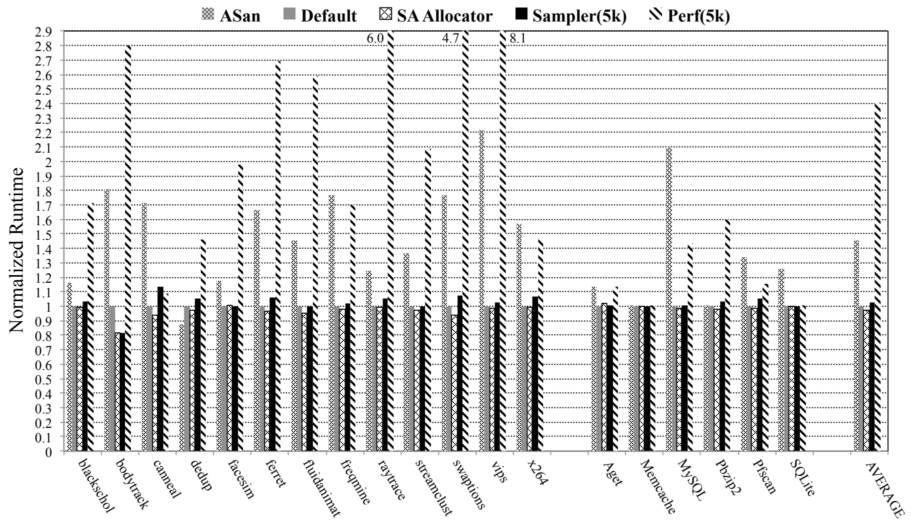

Fig. 7. Performance overhead of AddressSanitizer (ASan), `Sampler`'s allocator (SA Allocator), `Sampler`, and Perf when comparing to the default Linux libraries (Default).

configurations, and is easily tunable to achieve an optimal result. By default, `Sampler` utilizes a 16MB or 2048-object quarantine list.

VII. Related Work

A. Detecting Memory Errors

We focus on dynamic tools, and classify them as follows.

Dynamic Instrumentation: Numerous tools use dynamic instrumentation, including Valgrind's Memcheck tool [9], Dr. Memory [6], Purify [7], Intel Inspector [8], and Sun Discover [10]. Although these tools may use different dynamic instrumentation engines [9], [53], [54], they share an obvious advantage in that there is no need for recompilation or modification of programs. However, two serious shortcomings still prevent their adoption in production software: (1) they typically have very high performance overhead. For instance, Valgrind runs 20× slower [9], and Dr. Memory introduces around 10× runtime overhead [6]. (2) Normal users may still not have the expertise to use these tools.

Compiler-based Analysis and Instrumentation: Many tools utilize the compiler to perform static analysis at first, then instrument correspondingly to reduce the overhead [2], [11]–[15], [55], [56]. The state-of-the-art of this approach, AddressSanitizer [2], imposes around 73% performance overhead. However, this overhead is still too high to be employed in a production environment. Delta pointer detects both contiguous and non-contiguous overflows by carefully manipulating pointer arithmetic operations [56]. However, these tools cannot detect errors in code with no instrumentation in place.

Interposition: Multiple prior approaches use a mixture of library interposition and virtual memory techniques to detect memory errors [16], [19], [57]–[64]. DoubleTake [16] and iReplayer [20] employ the evidence-based approach to detect heap overflows and use-after-frees, where iReplayer surpasses DoubleTake by supporting multithreaded applications. These

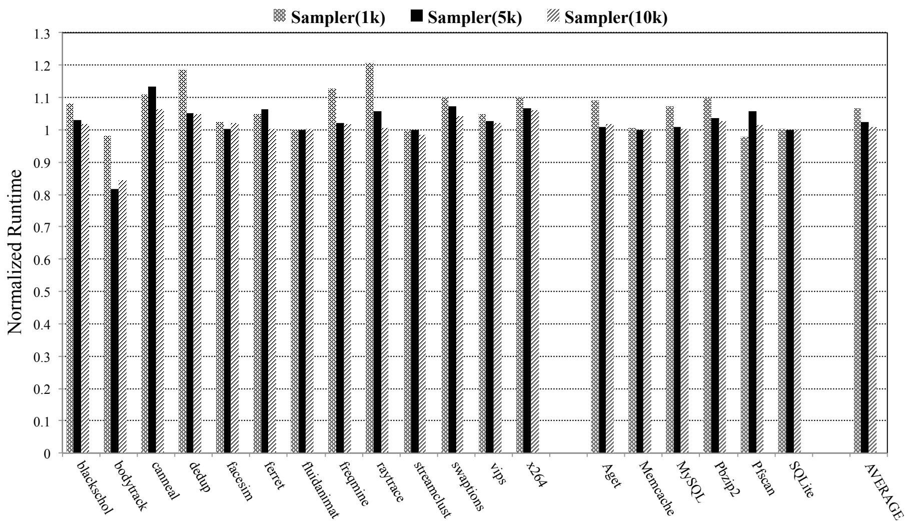

Fig. 8. Performance overhead for `Sampler` utilizing a series of increasing sampling periods.

	Memory Overhead (MB)			
Applications	ASan	Linux	SA Alloc.	`Sampler`
blackscholes	702	627	632	633
bodytrack	367	34	384	432
canneal	1933	963	1042	1721
dedup	1495	1486	2193	2326
facesim	3281	318	391	481
ferret	408	59	106	134
fluidanimate	401	400	415	423
freqmine	907	869	1081	2621
raytrace	1945	1162	1917	2247
streamcluster	184	114	118	123
swaptions	383	7	8	31
vips	360	32	33	61
x264	424	165	179	204
Aget	20	2747	4397	5923
Memcached	26	6	8	8
MySQL	302	106	152	173
Pbzip2	351	245	272	274
Pfscan	485	423	426	427
SQLite	464	9	14	77
Total	14438 (205%)	7028 (100%)	9376 (134%)	12402 (176%)

TABLE III
MEMORY OVERHEAD (IN MB) FOR APPLICATIONS USING ADDRESSSANITIZER, THE DEFAULT LINUX LIBRARIES, `SAMPLER`'S ALLOCATOR, AND `SAMPLER` (WITH A SAMPLING PERIOD OF 5000).

works implant canaries in the original executions, check for evidence upon the end of epochs, and detect root causes of memory errors using identical re-executions (when evidence of memory errors is discovered). They impose around 5% recording overhead for programs without memory errors. However, they only detect write-based failures, and cannot support multithreaded programs with ad-hoc synchronizations. `Sampler` detects both write-based and read-based errors with similar overhead, and supports all applications without any issue.

Page Protection: There is a body of work that leverages a page protection mechanism to achieve memory safety [65]–[69] at the expense of increased TLB pressure and performance overhead. The most efficient work, Oscar [66], optimizes virtual page management, but still imposes 40% performance overhead. Further, it only supports temporal memory safety, such as use-after-frees. In contrast, `Sampler` supports a broader spectrum of memory errors at a much lower cost.

B. *Sampling-Based Detection*

There exist some approaches that utilize a sampling-based technique to detect other software bugs, such as race conditions [4], [70], [71] and memory leaks [32], [72], [73]. However, none of them focus on the same memory errors as `Sampler`.

VIII. Conclusion

This paper proposes PMU-based sampling to detect buffer overflows and use-after-free errors, rather than validating every memory reference as in existing work. Further, `Sampler` utilizes a novel custom allocator to avoid false alarms and improve its performance. Based on our evaluation, `Sampler` only imposes less than 3% performance overhead, while detecting all known bugs within a reasonable number of executions. `Sampler` is the first work that simultaneously satisfies efficiency, precision, completeness, accuracy, and transparency, which makes it a good candidate for in-production use.

IX. Acknowledgement

We thank the anonymous reviewers for their invaluable feedback. This material is based upon work supported by the National Science Foundation under Award CCF-1566154, CCF-1823004, CNS-1750503, and CNS-1814430. The work is also supported by Mozilla Research Grant and Google/AMD Faculty Research Awards.

References

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. Washington, DC, USA: IEEE Computer Society, 2013. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.13

[2] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: a fast address sanity checker," in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342849

[3] S. Lu, W. Jiang, and Y. Zhou, "A study of interleaving coverage criteria," in *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, ser. ESEC-FSE companion '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1295014.1295034

[4] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng, "Racez: A lightweight and non-invasive race detection tool for production applications," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985848

[5] NIST. National vulnerability database. [Online]. Available: http://nvd.nist.gov/

[6] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2190025.2190067

[7] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *In Proc. of the Winter 1992 USENIX Conference*, 1991.

[8] Intel Corporation, "Intel inspector xe 2013," http://software.intel.com/en-us/intel-inspector-xe, 2012.

[9] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250746

[10] Oracle Corporation, "Sun memory error discovery tool (discover)," http://docs.oracle.com/cd/E18659_01/html/821-1784/gentextid-302.html.

[11] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors," in *Proceedings of the 18th conference on USENIX security symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855772

[12] Frank Ch. Eigler, *Mudflap: pointer use checking for C/C++*, http://gcc.fyxm.net/summit/2003/mudflap.pdf, Red Hat Inc., 2003.

[13] N. Hasabnis, A. Misra, and R. Sekar, "Light-weight bounds checking," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2259016.2259034

[14] G. C. N. Necula, M. Scott, and W. Westley, "Ccured: Type-safe retrofitting of legacy code," in *Proceedings of the Principles of Programming Languages*, 2002.

[15] parasoft Company, *Runtime Analysis and Memory Error Detection for C and C++*, http://www.parasoft.com/jsp/products/insure.jsp, 2013.

[16] T. Liu, C. Curtsinger, and E. D. Berger, "Doubletake: Fast and precise error detection via evidence-based dynamic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884784

[17] T. Ye, L. Zhang, L. Wang, and X. Li, "An empirical study on detecting and fixing buffer overflow bugs," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, April 2016.

[18] F. Qin, S. Lu, and Y. Zhou, "Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, ser. HPCA '05. Washington, DC, USA: IEEE Computer Society, 2005. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2005.29

[19] Q. Zeng, D. Wu, and P. Liu, "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*, ser. PLDI '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993541

[20] H. Liu, S. Silvestro, , W. Wang, C. Tian, and T. Liu, "ireplayer: In-situ and identical record-and-replay for multithreaded applications," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018.

[21] B. Lucia and L. Ceze, "Cooperative empirical failure avoidance for multithreaded programs," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451121

[22] W. Zhang, M. de Kruijf, A. Li, S. Lu, and K. Sankaralingam, "Conair: Featherweight concurrency bug recovery via single-threaded idempotent execution," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451129

[23] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou, "Triage: Diagnosing production run failures at the user's site," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294275

[24] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea, "Failure sketching: A technique for automated root cause diagnosis of in-production failures," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815412

[25] J. Callaham, "There are now 1.2 billion office users and 60 million office 365 commercial customers." [Online]. Available: https://www.windowscentral.com/there-are-now-12-billion-office-users-60-million-office-365-commercial-customers

[26] B. Kasikci, C. Zamfir, and G. Candea, "Racemob: Crowdsourced data race detection," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522736

[27] D. R. Hanson, "A portable storage management system for the icon programming language," *Softw., Pract. Exper.*, vol. 10, 1980.

[28] M. Itzkowitz, B. J. N. Wylie, C. Aoki, and N. Kosche, "Memory profiling using hardware counters," in *SC '03: Proc. of the 2003 ACM/IEEE Conf. on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003.

[29] B. R. Buck and J. K. Hollingsworth, "Data centric cache measurement on the Intel Itanium 2 processor," in *SC '04: Proc. of the 2004 ACM/IEEE Conf. on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004.

[30] X. Liu and J. M. Mellor-Crummey, "A data-centric profiler for parallel programs," in *Proc. of the 2013 ACM/IEEE Conference on Supercomputing*, Denver, CO, USA, 2013.

[31] X. Liu, K. Sharma, and J. Mellor-Crummey, "Arraytool: A lightweight profiler to guide array regrouping," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628102

[32] C. Jung, S. Lee, E. Raman, and S. Pande, "Automated memory leak detection for production use," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568311

[33] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti, "Laser: Light, accurate sharing detection and repair," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016.

[34] T. Liu and X. Liu, "Cheetah: Detecting false sharing efficiently and effectively," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO 2016. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2854038.2854039

[35] P. J. Drongowski, "Instruction-based sampling: A new performance analysis technique for AMD family 10h processors," http://developer.amd.com/Assets/AMD_IBS_paper_EN.pdf, November 2007, last accessed: Dec. 13, 2013.

[36] Intel Corporation, "Intel 64 and IA-32 architectures software developer's manual, Volume 3B: System programming guide, Part 2, Number 253669-032," June 2010.

[37] Vince Weaver, *Linux support for various PMUs*, http://web.eece.maine.edu/~vweaver/projects/perf_events/support.html, 2015.

[38] G. Novark and E. D. Berger, "DieHarder: securing the heap," in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866371

[39] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Cambridge, MA, Nov. 2000. [Online]. Available: citeseer.ist.psu.edu/berger00hoard.html

[40] D. Lea, "The GNU C library," http://www.gnu.org/software/libc/libc.html.

[41] P.-H. Kamp, "malloc (3) revisited." http://www-public.tem-tsp.eu/~thomas_g/research/biblio/2015/gidra15asplos-numagic.pdf.

[42] T. Liu and E. D. Berger, "Sheriff: precise detection and automatic mitigation of false sharing," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048070

[43] Y. Feng and E. D. Berger, "A locality-improving dynamic memory allocator," in *MSP '05: Proceedings of the 2005 workshop on Memory system performance*. New York, NY, USA: ACM Press, 2005.

[44] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, "Freeguard: A faster secure heap allocator," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017. [Online]. Available: http://doi.acm.org/10.1145/3133956.3133957

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

[46] "perf_event_open(2) - linux man page," http://linux.die.net/man/2/perf_event_open, perf_event_open - set up performance monitoring.

[47] T. Zhang, C. Jung, and D. Lee, "Prorace: Practical data race detection for production use," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037708

[48] M. Dmytrychenko, "Invariant tsc support," https://software.intel.com/en-us/forums/intel-isa-extensions/topic/280440, 2011.

[49] D. Gross, "TLS performance overhead and cost on gnu/linux," http://david-grs.github.io/tls_performance_overhead_cost_linux, 2016.

[50] J. Zhou, S. Silvestro, H. Liu, Y. Cai, and T. Liu, "Undead: Detecting and preventing deadlocks in production software," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155654

[51] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[52] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[53] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[54] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, ser. CGO '03. Washington, DC, USA: IEEE Computer Society, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=776261.776290

[55] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," in *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*, 2004.

[56] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida, "Delta pointers: Buffer overflow checks without the checks," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190553

[57] H. Ayguen and M. Eddington, "DUMA - Detect Unintended Memory Access," http://duma.sourceforge.net/.

[58] M. D. Bond and K. S. McKinley, "Bell: Bit-encoding online memory leak detection," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168866

[59] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336769

[60] Mac OS X Develper Library, "Enabling the malloc debugging features," https://developer.apple.com/library/mac/#documentation/performance/Conceptual/ManagingMemory/Articles/MallocDebug.html.

[61] G. Novark, E. D. Berger, and B. G. Zorn, "Exterminator: automatically correcting memory errors with high probability," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007)*. New York, NY, USA: ACM Press, 2007.

[62] B. Perens, "Electric Fence," http://perens.com/FreeSoftware/ElectricFence/.

[63] Microfocus, "Micro focus devpartner boundschecker," http://www.microfocus.com/store/devpartner/boundschecker.aspx, 2011.

[64] Q. Zeng, M. Zhao, and P. Liu, "Heaptherapy: An efficient end-to-end solution against heap buffer overflows," in *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '15. Washington, DC, USA: IEEE Computer Society, 2015. [Online]. Available: http://dx.doi.org/10.1109/DSN.2015.54

[65] W. Han, B. Joe, B. Lee, C. Song, and I. Shin, "Enhancing memory error detection for large-scale applications and fuzz testing."

[66] T. H. Dang, P. Maniatis, and D. Wagner, "Oscar: A practical page-permissions-based scheme for thwarting dangling pointers," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*. USENIX, 2017.

[67] P. Kedia, M. Costa, M. Parkinson, K. Vaswani, D. Vytiniotis, and A. Blankstein, "Simple, fast, and safe manual memory management," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2017.

[68] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn, "Archipelago: trading address space for reliability and security," *ACM Sigplan Notices*, vol. 43, 2008.

[69] D. Dhurjati and V. Adve, "Efficiently detecting all dangling pointer uses in production servers," in *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*. IEEE, 2006.

[70] M. D. Bond, K. E. Coons, and K. S. McKinley, "Pacer: proportional detection of data races," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806626

[71] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924954

[72] M. Hauswirth and T. M. Chilimbi, "Low-overhead memory leak detection using adaptive statistical profiling," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XI. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/1024393.1024412

[73] S. Lee, C. Jung, and S. Pande, "Detecting memory leaks through introspective dynamic behavior modelling using machine learning," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014.