CSOD: Context-Sensitive Overflow Detection

Hongyu Liu*, Sam Silvestro†, Xiaoyin Wang‡, Lide Duan§, and Tongping Liu¶
University of Texas at San Antonio, United States
*liuhyscc@gmail.com, †Sam.Silvestro@utsa.edu, ‡Xiaoyin.Wang@utsa.edu,
§Lide.Duan@utsa.edu, ¶Tongping.Liu@utsa.edu

***Abstract*—Buffer overflow is possibly the most well-known memory issue. It can cause erratic program behavior, such as incorrect outputs and crashes, and can be exploited to issue security attacks. Detecting buffer overflows has drawn significant research attention for almost three decades. However, the prevalence of security attacks due to buffer overflows indicates that existing tools are still not widely utilized in production environments, possibly due to their high performance overhead or limited effectiveness.**

This paper proposes CSOD, a buffer overflow detection tool designed for the production environment. CSOD proposes a novel context-sensitive overflow detection technique that can dynamically adjust its detection strategy based on the behavior of different allocation calling contexts, enabling it to effectively detect overflows in millions of objects via four hardware watchpoints. It can correctly report root causes of buffer over-writes and over-reads, without any additional manual effort. Furthermore, CSOD only introduces 6.7% performance overhead on average, which makes it appealing as an always-on approach for production software.

***Index Terms*—Overflow Detection, Memory Safety, Memory Vulnerabilities**

I. Introduction

Buffer overflow is an important class of memory bugs in C/C++ programs. Buffer overflows not only cause erratic program behavior, but also can be exploited for security attacks, such as data corruption, control-flow hijack, and information leakage [55]. For instance, the Heartbleed bug, a buffer over-read problem in the OpenSSL cryptography library, affected more than half-a-million Internet servers, resulting in the leakage of sensitive private data [2]. Despite various detection techniques, buffer overflows still widely exist in different in-production applications [40].

Buffer overflows are especially hard to be expunged during development phases, since many of them are only triggered with specific inputs, which are likely to be neglected in testing. On the other hand, static detection tools may generate numerous false positives, or have scalability issues for large programs [56], which significantly limits their adoption. Additionally, the pervasive use of multi-threaded programs imposes additional challenges for testing: it is even impossible to find all bugs that can be triggered by a single input, since some bugs are only exposed in one particular interleaving, and the number of interleavings is exponentially proportional to the number of statements [33]. Therefore, many bugs are inevitably leaked to the deployed environment, and there is a strong demand for tools that can detect buffer overflows in the production setting.

Unfortunately, existing dynamic detection tools often impose significant performance overhead that prevents their adoption in the production environment [15]. They typically perform bounds-checking at every memory access by invoking functions instrumented using different instrumentation techniques, such as dynamic instrumentation [42], [10], [23], [25], [46] or static instrumentation [49], [5], [21], [22], [41], [47], [52], [54]. Typically, Valgrind or Dr. Memory impose more than 10× performance overhead [42], [10]. AddressSanitizer, referred to as ASan in the remainder of this paper, is the state-of-the-art of this type, which still incurs more than 39% performance overhead despite many optimizations, such as static analysis, efficient shadow mapping, and encoding [52]. Additionally, ASan only detects problems caused by instrumented components, while skipping those caused by many non-instrumented libraries. Some detection tools introduce low runtime overhead, but with their own issues. DoubleTake [32] and iReplayer [31] only detect buffer over-writes due to their evidence-based mechanism, while leaving over-reads undetectable, e.g. Heartbleed [2], [17]. Control-Flow Integrity (CFI) can only detect overflows that cause unexpected execution flow, omitting buffer over-reads and overwrites that do not alter the flow [4]. Boud employs hardware watchpoints to achieve its low overhead, but only focuses on a small number of arrays within loops that are explicitly instrumented in the compilation phase [15].

In this paper, we propose a tool, called CSOD, that detects overflows using hardware watchpoints, but without the requirement of explicit instrumentation. CSOD places watchpoints just at the boundary of heap objects in order to detect all out-of-bounds reads and writes. The use of watchpoints balances overhead and effectiveness: (1) it can detect both read-based and write-based overflows; (2) A watchpoint is only fired when the watched address is accessed, thus it can report the precise calling context of an overflow, and will never report false alarms; (3) It does not impose additional runtime overhead, when there is no overflow.

However, there is a technical challenge caused by the limited number of hardware watchpoints, since there are only four available [60]. CSOD further proposes a novel calling-context-based detection due to the following key observation: *heap objects with the same allocation calling context typically have the same access behavior, determined by the program logic*. That is, a heap object accessed by given statements, and/or whether it has an overflow, is already determined by the program itself. Based on this insight, CSOD significantly reduces

its focus to the number of different allocation calling contexts inside a program, instead of a much larger number of heap objects. However, the number of active calling contexts can still be much larger than the number of watchpoints. CSOD further dynamically adjusts its strategy of placing watchpoints based on the monitored behavior: (1) an object whose calling context has a larger number of allocations will have a lower probability to be selected. This strategy ensures that each calling context may have a similar chance of being sampled. Also, this strategy favors objects from a calling context with fewer allocations, which shares a similar insight as existing work, e.g. SWAT [24], in that objects from a calling context with fewer allocations may have a higher chance of containing latent bugs. (2) CSOD decreases the possibility of watching objects from a calling context, if objects sharing this calling context were watched before but without observing overflows. More details are further discussed in Section III-B2.

Based on our evaluation of nine known bugs, including large programs with millions of lines of code (e.g. MySQL), CSOD successfully detected all overflows within few executions. For each execution, CSOD has a detection probability between 10% and 100%, with 58% on average. Note that although CSOD may miss a particular bug in a certain execution, it will catch this bug eventually with a sufficient number of executions. It is particularly suitable for the crowdsourcing or cloud environments, where a program will be executed repeatedly by a large number of users. Meanwhile, it only introduces 6.7% performance overhead on average, which is efficient enough to be employed in production environments. CSOD does not require any manual effort from users, and can report root causes of overflows without incurring any false alarms. Therefore, programmers can quickly identify problems based on the precise and accurate reports of CSOD, without the need for further confirmations. Overall, CSOD provides the following contributions:

- CSOD proposes a novel context-sensitive method to detect heap overflows with hardware watchpoints. This tool balances performance and effectiveness, and can be employed in production environments.
- Our extensive experiments confirm that CSOD detects overflows with a probability of 58% on average, with around 6.7% performance overhead.

The remainder of this paper is organized as follows. Section II first describes the background of watchpoints and the basic idea of CSOD. The detailed implementation is described in Section III. After that, an evidence-based optimization is further introduced in Section IV. Then, we evaluate CSOD in Section V, and discuss CSOD's weaknesses in Section VI. In the end, Section VII discusses related work, and Section VIII concludes the paper.

II. Overview

This section first introduces the background of hardware watchpoints, then describes the basic idea of CSOD.

A. Background of Watchpoints

Watchpoints, also known as "data points" [18] or "debug registers" [60], were designed for debugging purposes. Intel-based systems only support six debug registers. However, only four can be utilized to watch different linear addresses [60], while the remaining two are utilized to control debugging features or obtain the status. Watchpoints can be installed via system calls. Traditionally, `ptrace` is used to install them within the user space [27]. However, a separate process should be created for `ptrace` to install watchpoints, which incurs significant performance overhead due to communication between processes. Fortunately, Linux has another system call, `perf_event_open`, which allows installation within the same process [59].

B. Basic Idea of CSOD

Figure 1 shows the overview of CSOD. CSOD is a drop-in library that can be linked to applications via the "rdynamic" compilation flag or be preloaded by setting the *LD_PRELOAD* environment variable. CSOD includes six components: the Alloc/Dealloc Monitoring Unit, Sampling Management Unit, Watchpoint Management Unit, Signal Handling Unit, Termination Handling Unit, and Canary Management Unit. Among them, the last two components are only used for the evidence-based approach.

CSOD intercepts allocations and deallocations of applications via its Alloc/Dealloc Monitoring Unit. Upon every allocation, CSOD first contacts its Sampling Management Unit to determine whether to monitor this newly allocated object. If yes, CSOD relies on its Watchpoint Management Unit to choose one out of the four watchpoints, and installs it at the boundary of this object, as shown in Figure 2. Otherwise, CSOD simply updates the number of allocations, and adjusts the probability of the current calling context. After the installation, any memory read/write access on the watched linear addresses will trigger a *SIGTRAP* signal, which will be handled by the Signal Handling Unit. Inside the handler, CSOD reports the statement and its full calling context to the user. In addition, CSOD also reports the allocation calling context of the overflowing object, which can be obtained from the Watchpoint Management Unit. Upon each deallocation, CSOD checks whether this object is currently being watched. If so, CSOD removes the watchpoint on its corresponding address. Otherwise, CSOD does nothing upon deallocations.

III. Implementation Details

This section describes the detailed implementation of CSOD's major components that are shown in Figure 1.

A. Alloc/Dealloc Monitoring Unit

CSOD intercepts allocation and deallocation routines of applications, such as `malloc()` and `free()`, and handles them as follows. The interception is achieved by preloading memory allocation and deallocation routines, without the need to recompile or change applications manually.

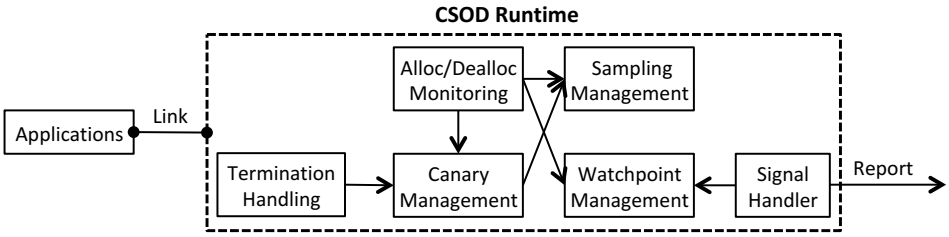

Fig. 1. Overview of CSOD

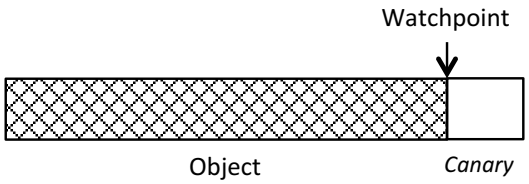

Fig. 2. Installation of watchpoints

1) Allocation: Upon every allocation, CSOD determines whether to install a watchpoint on this object. It first obtains the calling context of the current allocation, then obtains the probability of monitoring the current calling context from the Sampling Management Unit. After that, it determines whether a watchpoint should be installed on this object based on the probability and a randomly-generated number. For instance, given a probability of 10%, if a random number modulo 100 is less than 10, CSOD will install a watchpoint for this object, where the watchpoint is chosen via the replacement policy described in Section III-C2. Otherwise, this object will not be watched.

Identification of the calling context: Upon each allocation, CSOD obtains the calling context and requests its probability from the Sampling Management Unit, which is maintained in a hash table. However, obtaining and comparing the entire calling context is very expensive, especially for applications with a large number of allocations. CSOD utilizes the combination of the first level calling context above the CSOD library, e.g. the statement invoking the memory allocation, and the stack offset as the identifier of the calling context. If applications are compiled with the `-fno-omit-frame-pointer` flag, CSOD utilizes the built-in functions (e.g. `__builtin_return_address`) to obtain the next level of calling context above CSOD, instead of using the expensive `backtrace` function. Also, CSOD only acquires the whole calling context for the first time, with the use of the expensive `backtrace` function, when the combination cannot be found in the hash table. To our understanding, the chance that different calling contexts will have the same first-level calling context and stack offset will be extremely low, if not completely impossible. Furthermore, even if the first level calling context and the stack offset are the same as an existing one, this will not affect the detection correctness: CSOD can still correctly report the full calling context of a failure, since it only reports the context upon failure. However, CSOD may treat two different contexts as the same, which may affect the sampling probability. Also, CSOD may report the allocation calling context incorrectly.

Random number generator: The random number generator may significantly affect performance, since the generator is invoked upon every allocation. CSOD ports the random generator from OpenBSD's memory allocator, but changes it to support per-thread generation [37]. Currently, both OpenBSD and the `rand` function of `glibc` only support a global generator shared by multiple threads, and which further utilize a global lock to prevent race conditions, unnecessarily degrading the performance of multithreaded applications.

2) Deallocation: Upon every deallocation, CSOD checks whether the current object is being watched. If yes, the corresponding watchpoint will be removed. The detailed implementation for disabling watchpoints is discussed in Section III-C. Otherwise, no action will be required.

B. Sampling Management Unit

The Sampling Management Unit is an important component of CSOD, as it is responsible for adjusting the probabilities of different allocation calling contexts, based on the number of allocations and the watched times. It provides service to the Alloc/Dealloc Monitoring Unit and the Canary Management Unit.

1) Maintaining Probabilities for Calling Contexts: CSOD utilizes a global hash table to track the probabilities of different allocation calling contexts. As discussed above, the hash table utilizes the combination of two values as the key, the first-level calling context and the stack offset of the allocation. The size of the hash table is set to a large number to reduce hash conflicts. For all contexts that hash to the same value, a linked list is utilized to track these contexts, which has its own lock to guarantee correctness. Due to the fact that most applications may not have an excessive amount of distinct calling contexts, the hash table is expected to have very few conflicts, although at the cost of memory consumption.

2) Adaptively Adjusting Probability: As described above, the most critical challenge is to employ four watchpoints to effectively watch targets from thousands or millions of active heap objects. CSOD designs its sampling algorithms based on the calling contexts.

In CSOD, every calling context will be assigned a probability of 50% initially. This indicates that the calling context, without having been previously observed, is treated by CSOD as if it

were equally likely to either contain a bug or be bug-free. CSOD utilizes the following methods to adaptively adjust the probability for every calling context.

- **Degradation on each allocation:** After every allocation, regardless of whether the object is being watched, the probability of the calling context will be degraded by 0.001%. Therefore, an object from a calling context with a larger number of allocations will be made less likely to be watched each time.
- **Degradation after each watch:** CSOD reduces the probability of a calling context by half each time, if a calling context has been watched. Thus, objects from a calling context with fewer allocations have a higher chance of being observed.
- **Installation due to availability:** If a watchpoint is available, CSOD will watch an object, regardless of its probability. Thus, we never waste precious hardware watchpoints. This method also increases the detection effectiveness for the first few objects, which are more likely to be affected by input parameters.

Note that these percentages are pre-defined macros used at compilation time, which could be further adjusted based on the behavior of programs. However, based on our experiments, these numbers generally work well. Due to these adaptive methods, the probability of some calling contexts can be quickly degraded, specifically when there are a large number of allocations from this calling context, or objects from this calling context have been watched multiple times before. CSOD still maintains a lower bound on the probability in order to guarantee that every calling context has some chance to be watched in the future. Currently, the lower bound is set to 0.001%, and the probability will not be further reduced upon reaching this value.

Handling the probability with a large number of allocations: Some applications, such as Swaptions of PARSEC, may contain calling contexts with an extremely large number of allocations. Even with a probability of 0.001%, objects from such calling contexts can be watched too frequently, incurring substantial overhead associated with installing watchpoints. Thus, CSOD further reduces the probability of the calling context to 0.0001% when there are more than 5,000 allocations on this calling context within a preset time period (currently set to 10 seconds). After this time period has elapsed, the probability of objects from this calling context will again be increased to the lower bound.

C. Watchpoint Management Unit

The Watchpoint Management Unit is in charge of the installation, replacement, and removal of watchpoints.

1) Installing Watchpoints: The pseudo code for installing a watchpoint is presented in Figure 3. A thread installs the watchpoint for all threads, since there is no way to know which thread will cause an overflow later. The first step of the installation is to invoke perf_event_open to set up the watchpoint, which returns a file descriptor that can be used to enable or disable this watchpoint. Then, we configure

```
FOR_EACH_THREAD(ithread, aliveThreads) {
  struct perf_event_attr pe;
  pe.type = PERF_TYPE_BREAKPOINT;
  pe.bp_type = HW_BREAKPOINT_RW;
  pe.bp_addr = address;
  ......
  fd = perf_event_open(&pe, ithread->tid, -1, -1, 0);
  myflags = fcntl(fd, F_GETFL, 0);
  fcntl(fd, F_SETFL, myflags | O_ASYNC);
  // Send a SIGTRAP signal
  fcntl(fd, F_SETSIG, SIGTRAP);
  // Deliver it to the current thread
  fcntl(fd, F_SETOWN, ithread->tid);
  // Enable the watchpoint from now on
  ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);
}
```

Fig. 3. Pseudo code for installing a watchpoint

notifications to be asynchronous, the signal to be SIGTRAP, and specify the signal to be sent to the thread accessing the target address. Finally, we can enable the watchpoint on that thread. *It is very important to allow the current thread to handle the signal, otherwise there is no way to report the statement causing the current buffer overflow in multithreaded programs*. Before installing watchpoints, the signal handler should be set up correctly.

As shown in Figure 3, a watchpoint will be installed on all threads, using their thread ID number. Therefore, CSOD intercepts the pthread_create() invocation in order to acquire the thread ID for each thread. All alive threads are tracked in a global list, shown as aliveThreads in Figure 3.

2) Replacing Watchpoints: Upon every allocation, CSOD determines whether this object should be watched. If available watchpoints exist, they will be utilized first, without removing already-installed watchpoints. Otherwise, one installed watchpoint should be preempted. In order to avoid frequent replacements, CSOD only replaces a watchpoint when the probability of the new object is larger than that of the existing object. The probability of an existing object will be reduced when it has been installed for a long period of time (e.g., 10 seconds). This method is reasonable, since an object without overflows for an extended period will likely have a lower chance of experiencing overflows in the future. We have considered the following replacement policies, and evaluate their effectivenesses in Section V-A1.

Naive policy: The naive policy is a no-preemption policy: a watchpoint will be kept until its corresponding object is deallocated. Upon an object's deallocation, there is no need to watch further accesses on the object. Then, the watchpoint on this object will be removed in order to watch other addresses.

Random policy: When there are no available watchpoints, one out of four existing watchpoints will be chosen randomly to be removed. If the chosen one has a lower probability (dynamically changed due to the time) than the new one, then it is replaced by the new one. Otherwise, CSOD will proceed to

check the next watchpoint after this chosen one, until finding one with a lower probability.

Near-FIFO policy: The FIFO policy indicates that the first-installed watchpoint will also be the first to be replaced. To support this, a circular buffer is utilized to track four watchpoints, and a pointer is utilized to point to the first-installed watchpoint. However, as described before, a watchpoint will be removed upon deallocation. Therefore, our policy is called "near-FIFO" due to the balance between maintaining the FIFO sequence and minimizing performance overhead. In order to maintain the correct sequence after installation and deallocation, we should sort the circular buffer every time. However, these operations should be protected by a lock in a multi-thread environment to avoid data races. Instead, `CSOD` only updates the pointer to the next position, if the corresponding watchpoint has been replaced. This update operation can be performed correctly with an atomic instruction. Since deallocations may change the FIFO order, the replacement process is no longer in a strict FIFO order. Therefore, for this reason it is referred to here as the "near-FIFO" order.

3) Removing Watchpoints: Removing a watchpoint can be divided into two steps, as shown in Figure 4. The first step is to obtain the file descriptor associated with a given address. Every watched object maintains two addresses: one is the starting address of this object, and the other is the actual canary address that is being watched. At each object deallocation, the `free` routine is only provided with the starting address of this object, which will be utilized to determine whether it matches the address of any existing watchpoint. From this, we can obtain a watchpoint's corresponding file descriptor.

The second step is to remove the current watchpoint for all alive threads, which is the opposite operation of the installation. To remove a watchpoint from one thread, two system calls are invoked: one to disable the corresponding event, and the other to close the file descriptor, as shown in Figure 4.

```
watchObject* object = getObjectByAddr(addr);
FOR_EACH_THREAD(ithread, aliveThreads) {
  int fd = object->watchpoint_fd[ithread->index];
  ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
  close(fd);
}
```

Fig. 4. Pseudo code for removing a watchpoint

D. Signal Handler

When a thread accesses one of the watched addresses, the OS will send an asynchronous signal to the corresponding thread immediately, as discussed in Section III-C1. Inside the signal handler, `CSOD` should identify which watchpoint was fired, then report the overflow.

1) Identifying Fired Watchpoints: `CSOD` registers the signal handler via `sigaction`, and configures its signal handler to be of type `sa_sigaction`. Thus, it is possible for `CSOD` to obtain more information about the trap.

Within the signal handler, `CSOD` first obtains the file descriptor from the `siginfo_t` structure, which can be utilized to determine which watchpoint was fired in the signal handler. Since `CSOD` keeps all information associated with each watchpoint, such as the file descriptor, the watched address, and its allocation calling context, `CSOD` compares the current file descriptor with each of these saved file descriptors one-by-one, in order to identify which watchpoint has triggered the signal.

2) Reporting Buffer Overflows: `CSOD` reports two types of information to the user for each buffer overflow: the corresponding allocation calling context, and the overflowing site. The allocation calling contexts are kept in the hash table as described above. The overflowing calling context can be collected by analyzing the calling context of the signal handler. As described in Section III-C1, we have configured the signal to be delivered to the current thread that caused the buffer overflow. Thus, the overflowing thread will be stopped immediately upon accessing one of these watched addresses. `CSOD` simply employs the `backtrace` function to obtain the calling context of the overflowing site. It then invokes `addr2line` to print the detailed line number for each level of the calling context, if the symbol information was not stripped from the binary. Otherwise, `CSOD` will report the binary address of the corresponding statements.

IV. Optimizations

`CSOD` introduces two optimizations to further improve its effectiveness, described as follows.

A. Reviving Mechanism

It is possible that objects from one calling context do not have overflows across multiple watches, then suddenly one object from this context is overflowed due to a different input. In order to handle such situations, `CSOD` further introduces a **reviving mechanism** for those calling contexts with the minimum probability (0.001%): the probability of a calling context can be augmented randomly, by boosting it to 0.01% after a period of time. Therefore, `CSOD` partially handles the issues caused by different inputs.

B. Evidence-based Overflow Detection

Due to the limited number of watchpoints (four), `CSOD` may miss some overflows on each execution. To improve its effectiveness of detecting buffer over-writes, `CSOD` further borrows the evidence-based detection from existing work, such as DoubleTake [32] and HeapTherapy [63].

Upon each allocation, `CSOD` invokes the Canary Management Unit (shown in Figure 2) to implant a canary immediately after each heap object. The canary is a random value, as shown in Figure 5. `CSOD` also saves a pointer to the calling context so that it can be reported upon an overflow. The size of each object is also prepended to the metadata in order to identify the placement of the canary, and a pointer to the real object returned by the memory allocator is also added in order to support `memalign` operations. `CSOD` inserts an identifier to indicate the header of each object.

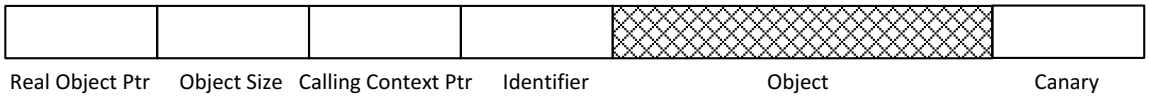

Fig. 5. Object layout with evidence-based detection

Upon every object deallocation, CSOD invokes the Canary Management Unit to check the integrity of canaries. Whenever a canary is found to be corrupted, a heap overflow has already occurred on the current object. CSOD immediately boosts the probability of the corresponding calling context to 100%, such that all following overflows sharing the same allocation calling context can be detected from then on.

CSOD also checks all canaries at the end of execution, via the registered exit function, which is controlled by its Termination Handling Unit shown in Figure 2. Since a program may crash due to overflows, or may never deallocate some objects in the case of memory leaks, CSOD registers a common signal handler to intercept erroneous exits caused by segmentation faults or aborts. At the end of the execution, all allocation calling contexts observed to have overflows are written to persistent storage as a file in order to detect buffer overflow in future executions.

V. Experimental Evaluation

We performed the experiments on a two-socket quiescent machine, where each socket has 8 cores with Intel® Xeon® CPU E5-2640 processors. It has 256GB main memory in total. The experiments were performed on Ubuntu 16.04, installed with Linux-4.4.25 kernel. We used GCC-4.9.1 with `-O2` and `-g` flags to compile all applications.

A. Effectiveness

The effectiveness evaluation was performed on 9 real applications with known heap-related buffer over-reads or over-writes, as shown in Table I. These buggy applications (and their corresponding buggy inputs) were obtained from Bugbench [34] or the CVE database. In addition, we specifically looked for some **large applications**, such as MySQL and Memcached. Among these bugs, three are buffer over-reads, including the notorious Heartbleed problem [2], as well as Libdwarf, and Zziplib. Other bugs are buffer over-writes. Each application contains only one known buffer overflow bug. For the Heartbleed vulnerability, we utilized Nginx-1.3.9 and OpenSSL-1.0.1f for the evaluation, as described in HeapTherapy [63].

Bug Reports: Upon detection, CSOD reports two types of information, including the calling context of the overflowing site, and its corresponding allocation calling context. An example bug report is shown as Figure 6, which shows the report for the Heartbleed problem. With the reported information, users can easily identify the problem, without the need for further confirmation.

TABLE I
Applications used for effectiveness evaluation

Application	Vulnerability	Reference
Gzip-1.2.4	Over-write	BugBench [34]
Heartbleed	Over-read	CVE-2014-0160 [19]
Libdwarf-20161021	Over-read	CVE-2016-9276 [50]
LibHX-3.4	Over-write	CVE-2010-2947 [11]
Libtiff-4.01	Over-write	CVE-2013-4243 [12]
Memcached-1.4.25	Over-write	CVE-2016-8706 [57]
MySQL-5.5.19	Over-write	CVE-2012-5612 [20]
Polymorph-0.4.0	Over-write	BugBench [34]
Zziplib-0.13.62	Over-read	CVE-2017-5974 [51]

```
A buffer over-read problem is detected at:
GLIBC/memcpy-sse2-unaligned.S:81
OPENSSL/ssl/t1_lib.c:2588
OPENSSL/ssl/s3_pkt.c:1095
……
NGINX/os/unix/ngx_process_cycle.c:138
NGINX/core/nginx.c:415

This object is allocated at:
OPENSSL/crypto/mem.c:312
OPENSSL/crypto/bn/bn_ctx.c:217
……
NGINX/http/ngx_http_request.c:577
NGINX/http/ngx_http_request.c:527
```

Fig. 6. Bug report for Heartbleed

1) Evaluation Results: To evaluate the effectiveness of CSOD, we ran each application 1,000 times, and Table II shows the number of executions in which the corresponding overflow bugs were detected. As described in Section III-C2, three watchpoint replacement policies were evaluated here, including the naive, random, and near-FIFO policies. The naive policy detected overflows in five simple applications with very few allocations, but could not detect bugs in complex programs with a large number of allocations. The near-FIFO policy and random policy had very similar results. Both detected bugs in a range between 10% and 100%, with an average of 58%.

Comparison with state-of-the-art: We also evaluated these applications using the state-of-the-art, ASan [52]. ASan cannot detect the overflows in Libtiff, LibHX, and Zziplib, when the corresponding libraries are not instrumented. For instance, the LibHX problem is actually caused inside the libHX.so library, as described in Figure 6. We also verify the false positives and false negatives of CSOD. **False positives:** CSOD does not introduce any false positives, since only

TABLE II
EFFECTIVENESS RESULTS FOR 1,000 EXECUTIONS

Application	Naive	Random	Near-FIFO
Gzip	1000	1000	1000
Heartbleed	0	364	396
Libdwarf	1000	480	459
LibHX	1000	929	885
Libtiff	1000	1000	1000
Memcached	0	163	183
Mysql	0	161	174
Polymorph	1000	1000	1000
Zziplib	0	110	102

accesses beyond object boundaries (actual overflows) will trigger watchpoints. **False negatives:** CSOD did not miss any overflows when considering the 1,000 executions together. However, CSOD may miss non-continuous overflows that skip boundaries.

TABLE III
DETAILED INFORMATION OF APPLICATIONS WITH BUGS

Application	Total Number		Number Before Overflow	
	Calling Context	Allocations	Calling Context	Allocations
Gzip	1	1	1	1
Heartbleed	307	5,403	273	5,392
Libdwarf	26	152	24	147
LibHX	4	5	1	1
Libtiff	1	1	1	1
Memcached	74	442	74	442
Mysql	488	57,464	445	57,356
Polymorph	1	1	1	1
Zziplib	13	17	13	17

What can affect detection effectiveness: As described above, the naive policy can always detect buffer overflows in some applications, such as LibHX, and Libtiff. To understand the reason, we further collected details of these applications, shown as Table III. The second and third columns show the total number of allocation calling contexts, and the number of allocations in these applications, while the last two columns show the number of occurrences of calling contexts and allocations before the overflowing object. Applications, including Gzip, Libdwarf, LibHX, Libtiff, and Polymorph, either have no more than four calling contexts, or the overflowing object appears within the first four allocations. Thus, the naive policy can always capture these bugs, since watchpoints are installed if available. However, it can never detect overflows for other applications, without preempting existing watchpoints or use of adaptive mechanisms. In contrast, CSOD can detect overflows with a probability between 10% and 36% for other applications, when using the random policy or near-FIFO policy. These results indicate that the adaptive mechanisms of CSOD make it effective at detecting overflows in large applications, with a large number of calling contexts and allocations. For instance, CSOD can detect the bug in MySQL about 174 times out of 1,000 executions, even if MySQL has 488 different calling contexts and 57,464 allocations. We observed similar results for Memcached and the Heartbleed problem as well.

2) Evidence-based Overflow Detection: We further evaluated whether the evidence-based overflow detection (described in Section IV-B) could enhance detection of buffer over-writes. Based on our evaluation of six buggy applications containing buffer overwrites (shown in Table I), *CSOD can always detect these over-write problems during their second execution, if missed in the first execution.*

B. Performance Overhead

We evaluated the performance of CSOD on the popular PARSEC benchmark suite [7], and several widely-utilized real applications, such as MySQL, Apache, Memcached, Aget, Pbzip2, and Pfscan. There are 19 multithreaded applications in total. We also evaluated the performance of ASan for comparison. *For a fair comparison, we only enabled the proper flags for ASan to detect heap buffer overflows, using minimally-sized redzones, although ASan can detect other issues as well.* Note that we did not instrument any external libraries. This indicates that the actual performance overhead of ASan could be much larger than the results shown here, if all libraries are instrumented.

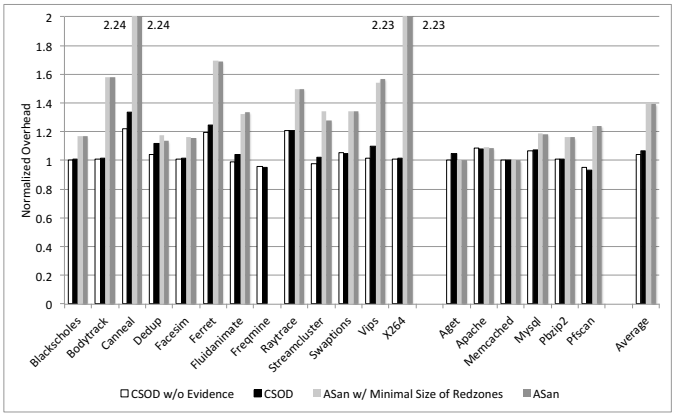

Fig. 7. Performance overhead of CSOD vs. ASan

PARSEC benchmarks were evaluated using the native inputs and 16 threads [7]. MySQL was tested via sysbench, with 16 clients and 100,000 maximum requests, where Figure 7 shows the throughput (e.g., requests per second). Memcached was evaluated using the python-memcached script [1], but with the number of loops increased to 20 iterations. Apache was tested by sending 100,000 requests via ab [3], and the number of requests per second is shown in Figure 7. Aget downloads a 600MB file from a quiescent server on the local network. Lastly, Pfscan performs a keyword search on 4GB data, while Pbzip2 compresses a 7GB file. The inputs are different from those used to evaluate effectiveness, as these inputs are appropriate for performance evaluation, similar to existing work. By contrast, for the effectiveness evaluation, those inputs were only used to trigger the bugs.

The results of CSOD and ASan are normalized to the performance of the default Linux system, as shown in Figure 7. Freqmine is omitted for ASan due to a program crash

in our evaluation environment. Overall, CSOD imposes 6.7% performance overhead on average, while ASan's overhead is around 39% [52]. Without evidence-based detection, CSOD only imposes 4.3% performance overhead (see "CSOD w/o Evidence"). Note that ASan's overhead can be higher when all libraries are instrumented, and ASan reports around 73% performance overhead in their paper [52].

The major overhead of CSOD comes from memory allocations: (1) it must obtain the corresponding calling context and its probability for each allocation. (2) It must determine whether it is necessary to watch this object, with the randomization support. (3) If the object is to be watched, CSOD needs to install a watchpoint for it, possibly by disabling another existing one. Also, CSOD introduces checking overhead at object deallocations, and at the end of execution, if evidence-based detection is enabled. Thus, CSOD's overhead is proportional to the number of allocations and the number of installed watchpoints. As described in Figure 3 and Figure 4, eight system calls are used to install and remove a watchpoint for each thread. We could further reduce the performance overhead by combining these system calls into one custom system call, but this requires modification of the underlying OS.

TABLE IV
CHARACTERISTICS OF APPLICATIONS

Application	LOC	CC	Allocations	WT
Blackscholes	479	4	4	4
Bodytrack	11,938	81	431,022	325
Canneal	4,530	10	30,728,172	79
Dedup	37,307	93	4,074,135	182
Facesim	45,748	109	4,746,070	369
Ferret	40,997	118	139,246	346
Fluidanimate	880	2	229,910	5
Freqmine	2,709	125	4255	218
Raytrace	36,871	63	45,037,327	561
Streamcluster	2043	21	8,861	30
Swaptions	1631	10	48,001,795	370
Vips	206,059	400	1,425,257	259
X264	33,817	60	35,753	37
Aget	1,205	14	46	16
Apache	269,126	56	357	27
Memcached	14,748	85	468	79
MySQL	1,290,401	1186	1,565,311	1,362
Pbzip2	12,108	13	57,746	58
Pfscan	1,091	6	6	5

To explain the performance overhead, we further collected different characteristics of these applications (as shown in Table IV), including lines of code, number of calling contexts of allocations ("CC" column), number of allocations ("Allocations" column), and number of watched times ("WT" column). Note that the input used for performance evaluation is different from that used for effectiveness evaluation. Because of this, the characteristics for the same applications are different.

As shown in Figure 7, "CSOD w/o Evidence" introduces more than 10% performance overhead for only three applications: Canneal, Ferret, and Raytrace. Canneal and Raytrace have large numbers of allocations – 30 million and 45 million allocations, respectively – where checking their contexts accounts for the majority of the overhead. Ferret runs for less than five seconds, which exaggerates the proportion of CSOD's initialization overhead. However, if the number of allocations is low, such as for Blackscholes or Aget, CSOD imposes negligible overhead.

In contrast, the major component of ASan's overhead comes from its checking of every memory access. ASan imposes little overhead for IO-bound applications, such as Aget or Pfscan. Also, ASan may impose less overhead if a large portion of time is spent in libraries without instrumentation, such as in Pbzip2. In general, ASan is more suitable for the development phase due to its large overhead, but not for the deployment phase.

C. Memory Overhead

We also evaluated the maximum memory overhead of CSOD using the same applications as those used in the performance evaluation. For server applications, including Apache, MySQL, and Memcached, we collected the physical memory consumption from the VmHWM field in the /proc/PID/status file. Memory consumption of other applications is collected from the output of the time utility. The maxresident field of this utility reports the maximum amount of physical memory consumed by an application [30].

TABLE V
MEMORY USAGE

Application	Original	CSOD		ASan	
		Kb	%	Kb	%
Blackscholes	613	630	103	673	110
Bodytrack	34	51	151	362	1079
Canneal	940	1,353	144	1,586	169
Dedup	1,599	1,781	111	1,530	96
Facesim	2,422	2,462	102	3,228	133
Ferret	68	90	133	413	610
Fluidanimate	408	434	106	489	120
Freqmine	1,241	1,262	102	-	-
Raytrace	1,135	1,306	115	2,523	222
Streamcluster	111	128	115	151	136
Swaptions	9	27	289	390	4178
Vips	59	78	133	333	570
X264	486	507	104	693	142
Aget	7	23	359	21	320
Apache	5	28	523	25	477
Memcached	7	26	391	24	359
Mysql	124	145	117	395	317
Pbzip2	128	148	116	411	322
Pfscan	4,044	3,688	91	4,142	102
Total	13,439	14,167	105	17,386	143

Table V shows the memory usage of the default library ("Original"), CSOD, and ASan. CSOD enabled its evidence-based overflow detection while collecting the memory data. ASan ran with the minimal size of redzones (16 bytes). Overall, CSOD introduces around 5% memory overhead in total, as compared to the default library. In contrast, ASan's memory overhead is around 43% in total, which is significantly higher than that of CSOD.

As shown in Table V, CSOD imposes a high memory overhead for small-footprint applications, such as Aget and Apache. Most overhead is caused by the addition of the 32-byte header prior to each object, as well as the 8-byte canary at the end of

each object, in order to support the evidence-based mechanism. However, CSOD only adds a single-digit percentage of memory overhead for large-footprint applications, such as Facesim and Pfscan. Overall, CSOD's memory overhead is acceptable for the production environment.

VI. Limitations

Due to its sampling mechanism, CSOD cannot detect all bugs using a single execution. However, as discussed in Section I, only a few executions are required to detect such bugs, and the probability of detection ranges from 10% to 100% for each execution. Additionally, CSOD reports root causes of overflows automatically and correctly, which will eliminate manual effort spent confirming software failures [26], [16]. CSOD is suitable for software with a large number of users, where multiple executions by multiple users help detect overflows.

CSOD also has the following limitations: (1) some objects are overflowed after a long period of time following their allocation. Due to the algorithms employed, the watchpoint may be preempted prior to the overflow occurring. However, the evidence-based detection will assuredly detect buffer over-writes, after evidence of these bugs is observed. (2) CSOD may not be able to detect non-continuous overflows that skip the addresses of installed watchpoints, since the watchpoints are only installed at the boundary of heap objects. (3) CSOD may have a low detection rate for overflows that are affected by different inputs during the execution. If objects from a calling context have been checked multiple times without encountering an overflow, the chance of observing future objects from the same calling context will be lowered.

CSOD cannot detect buffer overflows in global variables and stack variables. CSOD can only detect continuous overflows, which will always read or write the next word beyond the object's boundary, similar to DoubleTake [32]. ASan can detect overflows within redzones, regardless of stride or continuity, which is superior to CSOD. ASan cannot detect non-continuous overflows beyond the redzones.

VII. Related Work

This section discusses dynamic tools for detecting buffer overflows. Existing works can be classified by their instrumentation methods.

Dynamic instrumentation-based detectors: Numerous tools use dynamic instrumentation, including Valgrind's Memcheck tool [42], Dr. Memory [10], Purify [23], Intel Inspector [25], and Sun Discover [46], by using different dynamic instrumentation engines, such as Pin [35], Valgrind [42], and DynamiRIO [9]. These tools have an obvious advantage in that they are easy to use, since they do not require the recompilation or modification of programs. However, one major disadvantage of these tools is that they typically come with high performance overhead. For instance, programs running with Valgrind take 20× longer than the original execution [42]. Also, they are still unsuitable for normal users without expertise.

Static instrumentation-based detectors: Many tools perform static analysis to eliminate unnecessary instrumentation, which reduces performance overhead [49], [5], [21], [22], [39], [41], [47], [52], [54]. The state-of-the-art in this approach, ASan [52], imposes around 39% performance overhead in detecting buffer overflows, without instrumenting all libraries. This overhead is still too high to be employed in the production environment. Also, ASan can miss memory vulnerabilities contained in libraries [55], [61].

Hardware-assisted tools: Intel MPX [45] and CHERI [58] design new hardware to enforce memory safety. Intel MPX checks memory reads and writes by hardware instructions. It maintains pointer metadata in bounds tables, which is similar to SoftBound [39]. However, the high overhead is an obstacle to its adoption in practice. The CHERI architecture, compiler, and operating system support fine-grained, capability-based memory protection. However, programmers should modify the source code to leverage CHERI. Some existing tools utilize hardware watchpoints, but for different purposes. DataCollider uses hardware watchpoints to detect race conditions [18]. Kivati utilizes hardware watchpoints to detect atomicity violations with the assistance of static analysis [14]. Libson adopts hardware debug registers to detect whether a variadic function accesses memory outside the input argument list [29], which has a different target than that of CSOD. Boud employs hardware watchpoints to detect array bounds violations [15], which requires compiler-based instrumentation to insert canaries. One work concurrent with ours, Sampler [53], utilizes PMU-based memory access sampling to detect buffer overflows and use-after-frees, with similar overhead to that of CSOD. However, Sampler requires a custom memory allocator, and change of the underlying OS. In contrast, CSOD requires no availability of source code, no change of the underlying OS, and detects a larger range of buffer overflows in the range of all heap objects, with a similar performance overhead. These benefits come from CSOD's "Context-Sensitive" method that separates it with these existing works.

Interposition-based detectors: Many approaches use a mix of library interposition and virtual memory techniques to detect memory errors [6], [8], [13], [28], [36], [43], [44], [48], [62], [38], [63], [32], [31]. BoundsChecker interposes on Windows' heap library calls to detect memory leaks, use-after-free errors, and buffer overflows [38]. DoubleTake [32] and iReplyer [31] only introduce 4% performance overhead, but cannot detect read-based buffer overflows. Similar to DoubleTake, HeapTherapy implants canaries around each heap object and checks buffer over-writes upon memory deallocations [63]. HeapTherapy places non-readable/non-writable pages around every object in order to detect buffer over-reads, which has much higher overhead than CSOD.

VIII. Conclusion

This paper presents a context-sensitive approach to detect buffer overflows. A tool, called CSOD, utilizes this approach and four hardware watchpoints to detect buffer overflows in applications with up to millions of lines of code and millions of objects. CSOD can automatically report root causes of latent buffer overflows in both single-threaded and multi-threaded

applications. Overall, CSOD detects buffer overflows with a probability range of between 10% and 100%, while only imposing around 6.7% performance overhead. CSOD is suitable for production environments due to its low overhead, precise reporting, and no manual intervention.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Jingling Xue, and anonymous reviewers for their valuable suggestions and feedback. This material is based upon work supported by the National Science Foundation under Award CCF-1566154, CCF-1566158, CNS-1748109 and CCF-1823004. The work is also supported by Google Faculty Award, Mozilla Research Grant and the GREAT seed grant from UTSA Office of Vice President for Research, Economic Development and Knowledge Enterprise.

REFERENCES

[1] Pure python memcached client. https://pypi.python.org/pypi/python-memcached.

[2] *Heartbleed*, 2014.

[3] ab Developers. ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, 2005.

[5] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven 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*, SSYM'09, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association.

[6] Hayati Ayguen and Michael Eddington. DUMA - Detect Unintended Memory Access. http://duma.sourceforge.net/.

[7] Christian Bienia and Kai Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[8] Michael D. Bond and Kathryn 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*, ASPLOS XII, pages 61–72, New York, NY, USA, 2006. ACM.

[9] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[10] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 213–223, Washington, DC, USA, 2011. IEEE Computer Society.

[11] Bugzilla. "libhx: buffer overrun in hx_split()". https://bugzilla.redhat.com/show_bug.cgi?id=625866, 2010.

[12] Bugzilla. "libtiff (gif2tiff): possible heapbased buffer overflow in readgifimage()". http://bugzilla.maptools.org/show_bug.cgi?id=2451, 2013.

[13] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio 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*, ISSTA 2012, pages 133–143, New York, NY, USA, 2012. ACM.

[14] Lee Chew and David Lie. Kivati: Fast detection and prevention of atomicity violations. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 307–320, New York, NY, USA, 2010. ACM.

[15] Tzi-cker Chiueh. Fast bounds checking using debug register. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 99–113. Springer, 2008.

[16] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 820–831, New York, NY, USA, 2016. ACM.

[17] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

[18] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 151–162, Berkeley, CA, USA, 2010. USENIX Association.

[19] Exploit. "openssl heartbeat poc with starttls support". https://gist.github.com/takeshixx/10107280, 2014.

[20] Exploit. "mysql (linux) heap based overrun poc zeroday". http://www.exploit-db.com/exploits/23076/, 2017.

[21] Frank Ch. Eigler. *Mudflap: pointer use checking for C/C++*. Red Hat Inc., 2003.

[22] Niranjan Hasabnis, Ashish Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 135–144, New York, NY, USA, 2012. ACM.

[23] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[24] Matthias Hauswirth and Trishul 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*, ASPLOS XI, pages 156–164, New York, NY, USA, 2004. ACM.

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

[26] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 344–360, New York, NY, USA, 2015. ACM.

[27] Prasad Krishnan. Hardware breakpoint (or watchpoint) usage in linux kernel. *IBM Linux Technology Center, Canada*, pages 1–10, 2009.

[28] Doug Lea. The GNU C library. http://www.gnu.org/software/libc/libc.html.

[29] Wei Li and Tzi-cker Chiueh. Automated format string attack prevention for win32/x86 binaries. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 398–409. IEEE, 2007.

[30] Linux Comunity. *time - time a simple command or give resource usage*, 2015.

[31] Hongyu Liu, Sam Silvestro, and Tongping Liu. Idealreplay: Identical and efficient record-and-replay. Technical report.

[32] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Doubletake: Fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 911–922, New York, NY, USA, 2016. ACM.

[33] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. A study of interleaving coverage criteria. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 533–536, New York, NY, USA, 2007. ACM.

[34] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[35] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[36] Mac OS X Develper Library. Enabling the malloc debugging features. https://tinyurl.com/y6wszxad.

[37] David Mazieres. "arc4random(3)". https://www.openbsd.org, 1996.

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

[39] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 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 '09, pages 245–258, New York, NY, USA, 2009. ACM.

[40] National Institute of Standards and Technology. National vulnerability database. https://nvd.nist.gov/vuln/search/results?adv_search=true&form_type=advanced&results_type=overview&query=buffer+overflow.

[41] George C. Necula Necula, McPeak Scott, and Weimer Westley. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the Principles of Programming Languages*, pages 128–139, 2002.

[42] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[43] Gene Novark, Emery D. Berger, and Benjamin 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)*, pages 1–11, New York, NY, USA, 2007. ACM Press.

[44] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 397–407, New York, NY, USA, 2009. ACM.

[45] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(2):28:1–28:30, June 2018.

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

[47] parasoft Company. *Runtime Analysis and Memory Error Detection for C and C++*, 2013.

[48] Bruce Perens. Electric Fence. http://perens.com/FreeSoftware/ElectricFence/.

[49] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.

[50] Agostino Sarubbo. "libdwarf: heap-based buffer overflow in dwarf_get_aranges_list". https://goo.gl/mz9reR, 2016.

[51] Agostino Sarubbo. "zziplib: heap-based buffer overflow in __zzip_get32 (fetch.c)". https://goo.gl/yqtyrB, 2017.

[52] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[53] Sam Silvestro, Hongyu Liu, Tong Zhang, Changhee Jung, Dongyoon Lee, and Tongping Liu. Sampler: Pmu-based sampling to detect memory errors latent in production software. In *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51 '18, 2018.

[54] Y. Sui, D. Ye, Y. Su, and J. Xue. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. *IEEE Transactions on Reliability*, 65(4):1682–1699, Dec 2016.

[55] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[56] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[57] Talos. "memcached server sasl autentication remote code execution vulnerability". https://www.talosintelligence.com/reports/TALOS-2016-0221/, 2016.

[58] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, May 2015.

[59] Vincent M Weaver. Linux perf_event features and overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, volume 13, 2013.

[60] Wikipedia. x86 debug register. https://en.wikipedia.org/wiki/X86_debug_register.

[61] 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)*, pages 91–101, April 2016.

[62] Qiang Zeng, Dinghao Wu, and Peng 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*, PLDI '11, pages 367–377, New York, NY, USA, 2011. ACM.

[63] Qiang Zeng, Mingyi Zhao, and Peng 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*, DSN '15, pages 485–496, Washington, DC, USA, 2015. IEEE Computer Society.