Prober: Practically Defending Overflows with Page Protection

Hongyu Liu*†
liu2978@purdue.edu
Purdue University
USA

Ruiqin Tian*
rtian@email.wm.edu
William & Mary
USA

Bin Ren
bren@cs.wm.edu
William & Mary
USA

Tongping Liu†
tongping@umass.edu
University of Massachusetts Amherst
USA

ABSTRACT

Heap-based overflows are still not completely solved even after decades of research. This paper proposes Prober, a novel system aiming to detect and prevent heap overflows in the production environment. Prober leverages a key observation based on the analysis of dozens of real bugs: all heap overflows are related to arrays. Based on this observation, Prober only focuses on array-related heap objects, instead of all heap objects. Prober utilizes static analysis to label all susceptible call-stacks during the compilation, and then employs the page protection to detect any invalid accesses during the runtime. In addition to this, Prober integrates multiple existing methods together to ensure the efficiency of its detection. Overall, Prober introduces almost negligible performance overhead, with 1.5% on average. Prober not only stops possible attacks on time, but also reports the faulty instructions that could guide bug fixes. Prober is ready for deployment due to its effectiveness and low overhead.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**; **Software testing and debugging**.

KEYWORDS

Buffer overflow, Program analysis, Page protection

ACM Reference Format:
Hongyu Liu, Ruiqin Tian, Bin Ren, and Tongping Liu. 2020. Prober: Practically Defending Overflows with Page Protection. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416533

*Both authors contributed equally to this work.
†This work was initiated and partially conducted while Hongyu Liu and Tongping Liu were at the University of Texas at San Antonio.

1 INTRODUCTION

C/C++ applications are prone to memory errors, such as buffer overflows (including over-reads/over-writes). Buffer overflows will not only cause a program to crash, but also can be exploited to issue security attacks or cause information leakage [44]. Since it is not able to expunge all buffer overflows during development phases, highly depending on program inputs, significant research has been focused on detecting and preventing buffer overflows dynamically. Among them, stack-based overflows can be detected with very low overhead (less than 6.5%) via the shadow stack technique [44]. But heap-based overflows are still not solved yet, since they were still ranked as Top 2 vulnerabilities (as shown in Table 1).

Table 1: Top five vulnerabilities reported in 2018 [10].

Vulnerabilities	DoS	Code Execution	Overflow	XSS	Gain Information
16555	1852	3035	2492	2004	1426

Dynamic detection tools can be further divided into multiple types. The most common approach is to check the overflow before every memory access, which could stop the overflow immediately if a memory access is found to access red zones that are not supposed to be read or written. Existing work, such as Valgrind [30], Dr. Memory [5], and AddressSanitizer [40], employs this approach, but with static or dynamic instrumentation method, and different organization of red zones. However, even the state-of-the-art of this type, e.g. AddressSanitizer, still imposes over 40% performance overhead, in addition to its significant memory overhead. Therefore, this type of approach is only applicable for development phases, but not for the production environment.

Efficient approaches exist, such as Cruiser [48], DoubleTake [26], HeapTherapy [49], or iReplayer [24]. They detect buffer overflows after the effect, typically by checking the evidence of corrupted canaries. Although they impose very low overhead, generally less than 5%, they cannot detect read-based overflows because reads do not leave any evidence behind. Also, they cannot stop security attacks timely, since the detection may occur after exploits. Sampler also imposes little overhead by only checking sampled references via hardware performance counters [43]. However, Sampler cannot detect all overflows within one execution due to its sampling property, and shares the same issue that may only detect overflows after exploits.

We propose a novel system, called Prober, to overcome these issues. Prober has the following goals. First, Prober aims for in-production systems, which should impose low performance and memory overhead. Second, Prober should detect both read-based and write-based overflows. Third, Prober will stop overflows immediately, eliminating any possibility of memory exploits. Last but not least, Prober is able to report detailed information to assist bug fixes, e.g., allocation sites and faulty instructions.

To achieve these goals, Prober is based on **a key observation** that separates it from all existing work: *overflowing objects are typically related to arrays*. This observation is based on our analysis on dozens of bugs collected by existing work [47] (as further discussed in Section 2.1). We further confirmed that this observation holds for all overflows reported in a randomly-selected period in the CVE database. This observation is also aligned with the intuition: for an object not related to an array, there is no need of operating it with error-prone operations, such as pointer arithmetic instructions, string APIs, or loop operations, thus with a low possibility of overflows.

This key observation identifies the type of objects that may have buffer overflows, called as *array-related objects* or *susceptible objects*. Both terms will be utilized interchangeably in the remainder of this paper. To take advantage of this observation, *Prober proposes to separate array-related objects from normal objects, by placing them into a separate space*. Then Prober employs the page protection to detect overflows, an idea that was initially proposed by Electric Fence [37] but can be seamlessly integrated with this key observation that reduces the scope of detection. Prober allocates array-related objects from a heap that every object is separated from each other by protected pages. More specifically, Prober places every array-related object at the end of corresponding pages, while the next page will be set to be non-readable and non-writable (or protected). Therefore, any overflowing reference (either read or write) on the protected page will trigger a violation. By intercepting such violations, Prober will immediately stop the execution and any subsequent exploits, and report the faulty instructions precisely. Comparing to the mechanisms of using explicit checks [30, 40], page protection checks buffer overflows without actually checking every access, thus imposing no additional checking overhead other than the initial protection overhead. Prober is able to track all invalid accesses in the protected page, caused by either continuous or non-continuous overflows, which could potentially detect more issues than existing work using one word [24, 26, 43, 48, 49] or multiple words [40] as the canary.

However, the key challenge is to correctly identify all array-related heap objects. On the one hand, missing array-related objects will lead to no detection/protection of overflows caused by them, reducing the safety guarantee. On the other hand, if some unnecessary objects were included, it may impose some overhead unnecessarily. To this end, Prober proposes a **hybrid approach** to identify array-related objects. Some objects can be identified as array-related (or not) statically by analyzing the source code as described in Section 3.1, while the remaining ones will be identified in a hybrid way: Prober's static component (Prober-Static) identifies the basic type of such allocations (easier to do), instruments the size of such allocations with the compiler, and its runtime system (Prober-Dynamic) is responsible for determining whether it is an array-related object by the real allocation size. That is, if the size of an allocation is multiple times of the basic type, then this allocation site is identified as *susceptible allocation site*. Consequently, all future allocations from such sites will be allocated from the protected heap so that all overflowing references can be detected and prevented immediately.

In its implementation, Prober-Static relies on the LLVM compiler to perform the analysis and instrumentation at the Intermediate Representation (IR) level. Prober proposes to identify array-related allocations based on the allocation function, the definition of the `size` parameter, and the operations of the corresponding object. After that, Prober-Static further labels array-related allocation sites with simple instrumentation, so that Prober-Dynamic will place the corresponding objects in the protected heap. For objects that cannot be identified as array-related ones statically, Prober-Static simply labels the unit size so that Prober-Dynamic can determine its type dynamically. Overall, Prober is over-estimated so that it will not miss any array-related allocations.

Prober-Dynamic intercepts all memory allocations and deallocations so that it can determine array-related allocations and manage array-related allocations correspondingly. Prober's key observation restricts its protection scope to a small portion of objects, instead of monitoring all heap objects, which is one major reason why Prober runs much more efficiently than Electric Fence [37]. Further, Prober also implements carefully to reduce the overhead as follows: (1) It employs per-thread heaps to cache available/freed objects locally in order to reduce the contention among different threads, an idea borrowed from Hoard [3]; (2) It employs an information-computable design to reduce the checking overhead upon deallocations; (3) Freed objects are organized by the size of power-of-two pages in order to encourage the re-utilization of objects, without coalescence and splitting, which is different from Electric Fence [37].

We have performed extensive experiments to evaluate the performance overhead, memory overhead, and effectiveness. Based on the evaluation of 18 applications, Prober imposes only 1.5% performance overhead on average and around 25.9% memory overhead, making it applicable for in-production systems. To ensure that Prober does not miss any necessary instrumentation, we have confirmed that Prober instruments correctly for all known overflows collected by existing work [47]. Also, we further confirmed that Prober correctly detects and prevents 10 known overflows within real applications. Prober is ready for in-production systems due to its low overhead, timely prevention, and effectiveness.

Overall, this paper makes the following contributions.

- It makes a novel key observation that only array-related objects are prone to overflows based on our analysis of massive bugs. We further empirically confirm that this observation holds for randomly-chosen real bugs in the CWE database.
- It proposes a hybrid mechanism that ensures to identify all array-related allocations. Such a mechanism is based on the allocation function, the definition of `size` parameter and the operations on the corresponding object, or the combination of the unit size and the requested size.
- It designs and implements a new allocator to manage the protected heap efficiently, by borrowing multiple mechanisms originated from different memory allocators.
- The paper performs extensive evaluation on the performance and effectiveness of Prober, showing that Prober has the potential to be actually employed in the deployment environment.

Table 2: Analysis on 48 heap overflows collected by [47].

Type	Overflow Reason		Num(#)
Sub-structure overflows	Pointer arithmetic		0
	Loop operation		4
	System call		0
	String API	memcpy	2
		strncpy	3
		strcpy	1
Whole-structure Overflows	Pointer arithmetic		3
	Loop operation		20
	System call		2
	String API	memcpy	6
		strncpy	1
		strncmp	1
		memset	3
		sprintf	1
		memmove	1

The remainder of this paper is organized as follows. Section 2 first describes the key observation, the basic idea of Prober, and then describes the attack model of Prober. The detailed implementation is further described in Section 3, and the evaluation is presented in Section 4. After that, we discuss Prober's weaknesses in Section 5. In the end, Section 6 discusses related work, and Section 7 concludes.

2 OVERVIEW

This section first analyzes overflow bugs collected by an existing study [47], and derives our *key observation*: overflowing objects are all related to arrays. Based on this key observation, it further discusses the basic design and key challenges of Prober.

2.1 Observations on Heap Overflows

One recent work studies 100 "randomly selected bugs within the buffer overflow category from the CVE website" [47]. Based on their description, the study is objective due to random selection, representing the real situation of buffer overflows. Therefore, our analysis was based on these bugs to avoid any bias. Based on our analysis, these 100 overflow bugs include 48 heap overflows, and 52 stack or global buffer overflows. This section focuses on 48 heap overflows, as shown in Table 2. We have the following observations.

The first observation is that all of the heap overflows are involved with arrays, either sub-structure or whole-structure overflows. Here, a *whole-structure overflow* is an overflow that its allocation is an array of structures or basic units (e.g., characters, integers, or words). A *sub-structure overflow* is that the object (or allocation) itself is not an array, but the corresponding structure includes one or multiple arrays internally. It is intuitive that array-related objects are prone to overflows. If an allocation is just a structure, every field can be manipulated with a member access operator (e.g., "->" or "."), which should not cause the overflow. On the other hand, if an object is related to an array, then it is very likely to employ error-prone operations, such as pointer arithmetic instructions, string APIs, or loop operations.

The second observation is that whole-structure overflows are much more common than sub-structure overflows, consisting of around 79.2% of these bugs (with 38 bugs in total).

The third observation is that overflow bugs can be caused by multiple operations, such as pointer arithmetic instructions, string APIs, loop operations, or system calls, as further shown in Table 2. More specifically, 24 out of 48 overflows are related to loops during the iterations, and 19 overflows are related to string APIs. For instance, the `memcpy` function copies more memory than it should. These two categories actually consist of more than 89.5% of these bugs. In addition to these two categories, three overflows are related to pointer arithmetic, and two overflows occur when the `read` system call does not check the boundary of the buffer. Thus, overflow occurs if programs utilize the pointers to access the entry of an array, but without correctly checking its size.

Table 3: Heap overflows between 11/01/2018 and 02/15/2019.

Type	Overflow Reason		Num(#)
Sub-structure overflows	Loop operation		4
	String API	sprintf	1
Whole-structure Overflows	Pointer arithmetic		5
	Loop operation		15
	System call		1
	String API	memcpy	4
		strncat	1
		strncpy	1
		memset	2
		snprintf	1
		memmove	2

Confirming Key Observation: In order to further confirm our key observation, we further examined 65 heap overflow bugs reported in the National Vulnerability Database, with the published date between 11/01/2018 to 02/15/2019. Since only 37 bugs out of 65 bugs have a detailed description or have the source code information, we focused on these 37 bugs. Based on our analysis, all of these 37 bugs are array-related, where whole-structure overflows are still the most common types of overflows, with the percentage of 86.4% and a total of 32 bugs.

2.2 Basic Idea of Prober

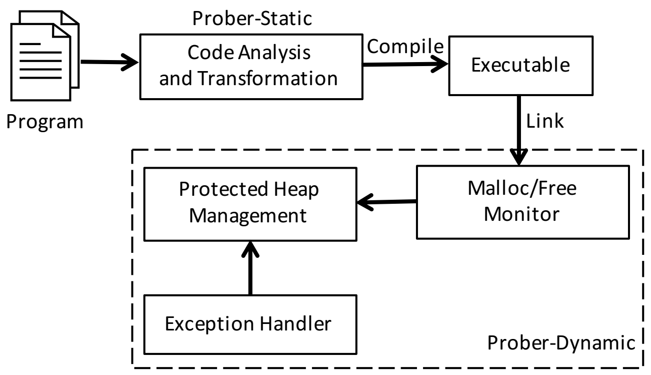

Figure 1: Overview of Prober.

Based on the key observation, Prober focuses only on array-related whole-structure overflows, where around 80% reported heap overflows to belong to. Since array-related objects are only a small percentage of all heap objects, the detection overhead can be dramatically reduced as further evaluated in Section 4 when using the page-protection mechanism. Prober does not handle sub-structure overflows in this paper, which will be the future work.

The design of Prober is illustrated as Figure 1. Basically, Prober includes two components, Prober-Static and Prober-Dynamic. Prober-Static is a static compile-time based tool that identifies and labels susceptible memory allocation sites, while Prober-Dynamic performs overflow detection/prevention and determines some array-related allocations on top of the static instrumentation.

2.2.1 Prober-Static. Prober-Static performs analysis and instrumentation at the Intermediate Representation (IR) level because of multiple benefits. First, LLVM IR offers multiple built-in functions that can facilitate the analysis and instrumentation. For example, *define-use* and *use-define* chains that track the definition and usage of memory allocation, can help determine an array-related allocation. Second, the analysis and instrumentation algorithm on LLVM IR is more robust, because many complicated cases (e.g., various macros) at the source code are simplified or merged at the IR level. Third, instrumenting at IR level provides the flexibility of registering the new code transformation pass in an appropriate position of the compilation chain, thus avoiding the possible side-effects to the subsequent analysis and code optimizations (e.g., loop optimizations) that are crucial to the code performance. Prober-Static analyzes the IR to determine array-related allocations, and marks susceptible allocation sites via the explicit instrumentation. Currently, Prober-Static is registered as a Link Time Optimization (LTO) pass so that it can handle definitions and usages located in multiple C/C++ files.

Research Challenges: The aim of Prober-Static is to design a robust compile-time analysis, which further includes two challenges. First, how to identify memory allocations, given that memory allocations have various forms, e.g., wrapper functions, or function pointers? Second, how to identify array-related memory allocations? Basically, Prober designs a hybrid mechanism to ensure correctness and completeness. If an allocation site can be identified statically, as described in Section 3.1, then it will be labeled explicitly. Otherwise, Prober-Static labels the size of its basic unit, and then relies on its dynamic component to determine array-related allocations.

2.2.2 Prober-Dynamic. Prober-Dynamic is a dynamic library that applications should be linked with. It intercepts all heap allocations and deallocations via its "Alloc/Free Monitor" module, and handles the allocation and detection for array-related objects. For objects allocated from array-related allocation sites, if Prober-static could identify them statically, Prober-Dynamic allocates these objects from a separate heap via its "Protected Heap Management" module. Basically, these susceptible objects will be separated by protected pages. In particular, they will be placed to the end of corresponding pages, with the next page as protected pages. Therefore, any overflow will be forced to land on protected pages, triggering the protection violation consequently. Prober-Dynamic has another component—"Exception Handler"–to deal with protection violations. Inside the exception handler, Prober precisely pinpoints the faulty instruction that causes the overflow by simply analyzing its calling context of exception. Then Prober stops the execution immediately, preventing any further exploits of overflows. Prober can be configured to detect out-of-one-page overflows easily. Note that Prober cannot detect overflows that do not access protected pages. This indicates that Prober cannot detect less-than-one-word overflows. However, this is not a real issue, since most heap allocators will return a word-aligned address. Less-than-one-word heap overflows practically will not cause any issue. Prober cannot detect underflows landing on the same page as the starting address of special objects, but will tolerate them instead.

Another task of Prober-Dynamic is to identify array-related objects, when they cannot be identified statically. It utilizes a simple mechanism to determine this dynamically: whether the requested allocation is multiple times of its basic structure. If that is the case, Prober-Dynamic will treat the allocation site as array-related objects, and follows the above description.

Research Challenges: Page-based protection guarantees that it generates no false positives, since memory references on the protected pages are guaranteed to be real overflows. However, the challenge is to design a system that could manage protected objects efficiently, since a naive method as Electric Fence imposes too much overhead to be employed in the deployment environment. Section 3.2.2 presents multiple mechanisms to reduce the contention and possible cache misses.

2.3 Attack Model

Prober targets to detect both read-based and write-based heap buffer overflows, and then stop any possible exploits immediately, based on explicit instrumentation. It utilizes the page-based protection to detect invalid accesses, which is available on any hardware that supports the virtual memory mechanism. Prober does not rely on a specific Operating System, which will be a general solution, although the current prototype is only implemented on top of Linux. Prober does not rely on any randomization mechanism in user space or kernel space. Prober could still work effectively, even if the hacker knows the source code of the application and Prober.

3 DESIGN AND IMPLEMENTATION

This section describes the detailed design and implementation of Prober that consists of two components, static instrumentation (Section 3.1) and runtime system (Section 3.2).

3.1 Compiler Analysis and Instrumentation

Prober-Static performs its static analysis and instrumentation on LLVM IR to identify all susceptible allocations, and relies on dynamic confirmation to confirm those ones that cannot be determined statically. Overall, our hybrid approach guarantees a 100% coverage for array-related allocations, which is over-estimated in reality. Prober-Static is implemented as one Link Time Optimization (LTO) pass because of two major considerations. First, the allocation function may be located inside a wrapper function, but this wrapper function is invoked in another C file, so an inter-module analysis (provided by LTO) is required. Second, placing instrumentation at link-time can effectively avoid complicating or interfering performance-critical compile-time analysis and optimizations (e.g., varied loop optimizations). Prober-Static determines array-related allocations in three steps, as further described in Section 3.1.1.

3.1.1 *Identify Susceptible Allocations.*

Prober-Static analyzes susceptible (or array-related) allocations in the following steps.

Step-I: Identify memory allocation functions: Based on our knowledge, memory allocations are invoked by several APIs and operators in C/C++, such as *new*, *malloc*(), *calloc*(), *realloc*(), *valloc*, *posix_memalign*(), and *memalign*(). But there are multiple situations as described in the following.

Basic Case: Some memory allocation invocations can be directly recognized according to the name in LLVM IR. For example, the `new[]` keyword is translated to `_Znam` in LLVM IR. Similarly, various macro definitions can also be recognized directly in IR level, because they have already been replaced by the preprocessor before being converted to IR.

Special Cases: Prober-Static also handles two more sophisticated but common cases. First, memory allocation is invoked inside a wrapper. For this case, Prober-Static recursively treats all functions in its calling stack as wrappers of memory allocation functions. Second, memory allocation is defined as a function pointer. Fortunately, LLVM translates function pointer calls to indirect calls in its IR, and the function invocation is specified by a load instruction. Listing 1 shows a simple example. The definition of `malloc_ptr` requires an additional check to determine whether line 2 is a memory allocation.

Listing 1: Memory alloc is defined and called as a fun ptr.

```
1   %4 = load i8* (i64)*, i8* (i64)**@malloc_ptr, align 8
2   %5 = tail call i8* %4(i64 %0)
```

Step-II: Identify array-related allocations: Prober-Static further identifies array-related allocations by the name of functions, the definition of allocation size, and the operations of the corresponding object. Table 4 lists multiple examples that cover 36 bugs analyzed in Section 2.1. The details of these examples are discussed as follows.

Type I can be identified by the name of memory allocation functions. For example, `new[]` is known as an operator to allocate an array, and `calloc` allocates an array with multiple objects with the same size. 5 out of 36 cases belong to this simple type.

Type II, III, and IV can be identified by the definition of `size` parameter. If its `size` parameter is defined (or manipulated) by `multiplication`, `addition`, and `strlen` operations, then the corresponding allocation is array-related. We can easily understand this by checking its contradiction. If an allocation is just for a single structure, a `sizeof` operation will be used to compute the `size` parameter, without these operations. Prober-Static employs LLVM's built-in `def-use` and `use-def` chains to assist the analysis on the definition of `size` parameter. 24 out of 36 cases can be analyzed using this method.

Type V can be identified by the operation on corresponding objects. As we know, some APIs, such as `read`, `fread`, `pread`, `readv`, read multiple bytes from the network or a file to the local buffer. Therefore, whenever one object appears as the destination buffer of these system calls, it should be tracked. Based on our analysis, 2 out of 36 cases belong to this type. Similarly, the analysis also requires the support of LLVM's built-in `def-use` and `use-def` analysis.

Table 4: Examples of susceptible allocations.

Type	Example	Explanation	Count

I	... = (**int***) new[5];
	... = (**int***) calloc(5,**sizeof**(**int**));
		Memory allocation calls `new[]` or `calloc`.	5

II	size = num * **sizeof**(**struct S**);
	... = (**struct S***) malloc(size);
		`size` is defined by a `multiply` operation.	13

III	size = size1 + size2;
	... = (**struct S***) malloc(size);
		`size` is defined by a add operation.	10

IV	size = strlen(buffer);
	... = (**int***) malloc(size);
		`size` is a return value of `strlen()`.	1

V	buffer = malloc(size);
	read(buffer, 0, size);
		Object is operated by array-related syscalls.	2

VI	... = (**int***) malloc(const_value);	`size` is a constant.	3

VII	size = (i > 0 ? **sizeof**(**int**) : 10 * **sizeof**(**int**));
	... = (**int***) malloc(size);
		`size` is from a branch that is potentially array-related.	2

SUM			36

Type VI requires further analysis, when the `size` parameter of an allocation is a constant integer. For most cases, if the `size` parameter is a constant, the corresponding allocation is an array. But there are some exceptions when analyzing in IR level. For instance, if a statement is like this, $(structS*)malloc(sizeof(structS))$, the `size` parameter is also interpreted as a constant integer in IR level. But this is not an array. To avoid the misidentification, Prober-Static further confirms whether the `size` is equal to the size of the corresponding data type. Although LLVM has some built-in functions to get the size of the object type, it requires some additional analysis to determine the object type. The challenge is to determine this when an allocation returns a void type pointer. Prober-Static adopts a `def-use` or `use-def` analysis to find the definition or the usage of the return value to figure out the object type.

Type VII is more complicated, since the object can be an array in some branches. More specifically, LLVM-IR represents these branches with a PHINode instruction. Prober-Static tracks all incoming values of this PHINode instruction. If at least one value belongs to **Type II, III, or IV**, this allocation is treated as array-related conservatively.

After the above analysis, Prober-Static will determine most allocations array-related or not and selectively protect the arrays and ignore the ones that are not array-related.

Step-III: Identify the object type (and unit size) for memory allocations non-determined: If a statement cannot be determined array-related or not in Step-II, Prober-Static labels the allocated object type (and thus the unit size) so that this allocation can be determined dynamically by Prober-Dynamic. Prober-Dynamic collects the size of an allocation size and divides it by the unit size[1]. If this result is greater than one, Prober-Dynamic will protect this memory allocation.

Prober-Static mainly employs LLVM's built-in `def-use` chains to find an object's type in its usage site. Prober-Static also relies on the metadata in LLVM IR to find the type information. Listing 2 and 3 show two examples of finding the object type with `def-use` chains

[1]A memory allocation might be used in more than one data types

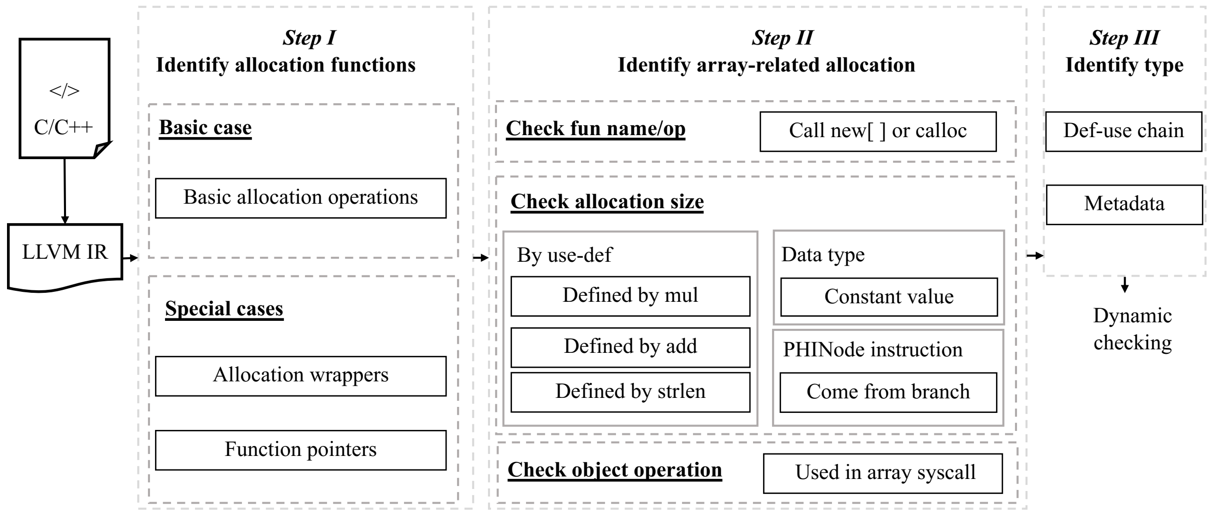

Figure 2: Identify susceptible allocations.

and metadata, respectively. Listing 2 illustrates that an explicit casting operation reveals the object type.

Listing 2: Identify the object type with a casting

```
%1 = call noalias i8* @malloc(i64 70) #4
%2 = bitcast i8* %1 to %struct.s*
```

Sometimes, it is difficult to find any obvious usage for a memory allocation, then the metadata information showed in Listing 3 also helps to find its object type.

Listing 3: Identify the object type with metadata

```
%21 = tail call i32 @mbuffer_create(%struct.mbuffer_t* nonnull %20,
    ↪ i64 %19) #7, !dbg !1409
!22 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
!1393 = !DILocalVariable(name: "r", scope: !1387, file: !137, line: 312,
    ↪ type: !22)
```

It is worth noticing that Prober-Static sets the type size as "1" by default, so even it cannot determine the object type statically via the above analysis, the allocation site will be protected effectively during the runtime. That is, Prober always ensures over-protection.

Put them together: Figure 2 summarizes Prober-Static's implementation. In Step-I, it checks each LLVM IR instruction according to one basic case and two special cases to identify all invocations of allocation functions, allocation function wrappers, and allocation function pointers. In Step-II, only these allocation invocations are further identified based on the following order. First, it checks the function name and operator. Second, it checks the allocation size with use-def, data type, and PHINode instruction information. Finally, it checks whether the operations of the corresponding objects is related to some special system calls. If an allocation meets any of these cases, then it is array-related. For them, Prober-Static label it explicitly as shown in Section 3.1.2. Otherwise, Prober-Static finds the object type (and type size) with either `def-use` chains or metadata in LLVM IR, and instruments the size of the allocation before the allocation so that Prober-Dynamic will confirm it dynamically.

In real-world applications, pointers and alias variables may complicate this analysis in two aspects. First, an alias pointer points to the protected allocation. However, this will not cause any issue, since Prober detects any access on the protected pages, no matter whether they are accessed via an alias or not. Second, an allocation function contains pointers or alias variables as its size parameter. Prober-Static relies on LLVM's pointer and alias analysis functions to associate these pointers or alias variables to the actual size variable and then performs further analysis. The evaluation in Section 4 demonstrates that Prober-Static can successfully identify and instrument array-related allocations for 46 bugs.

Listing 4: A LLVM-IR instrumentation example with *new*.

```
@specialMalloc = external thread_local global i8, align 1
define dso_local i32 @main() #0 {
  store volatile i8 −1, i8* @specialMalloc, align 1
  %6 = call i8* @_Znam(i64 20) #2
  ret i32 0
}
```

Listing 5: Equivalent C instrumentation of the *new* example.

```
extern __thread volatile bool specialMalloc;
int main(){
    specialMalloc = −1;
    int* b = new int[5];
    return 0;
}
```

3.1.2 *LLVM-IR Instrumentation.*

After a susceptible allocation site has been identified, a thread-local variable, e.g., *specialMalloc*, will be inserted to mark this site as a susceptible allocation. Here, the *specialMalloc* variable is an integer variable, with the value of "0" by default. This variable is set to "-1" before the allocation site if the allocation is array-related allocation. For instance, the *new* example of **Type I** in Table 4 is instrumented as Listing 4, where Listing 5 shows its equivalent C code for clarification. If a memory allocation is non-determinable statically, *specialMalloc* will be set as the size of the object type. Prober's runtime will determine if it should be protected.

3.2 Runtime System

As described in Section 2, the runtime system intercepts all memory allocations and deallocations so that all susceptible objects can be protected correspondingly. Therefore, the runtime system includes multiple components, such as malloc/free monitor (Section 3.2.1), protected heap management (Section 3.2.2), and exception handler (Section 3.2.3), as further shown in Figure 1.

3.2.1 Malloc/Free Monitor. Prober intercepts all memory allocations and deallocations with the preloading mechanism. Prober determines whether an object should be allocated from the protected heap upon memory allocations, and whether to return an object to the protected heap upon deallocations.

Prober relies on the static instrumentation to determine an array-related object. As described in Section 3.1.1, a thread-local variable (*specialMalloc*) will be labeled by the Prober-Static: If an object is identified as not-array related statically, with the value 0, the object will be allocated from the default allocator; If the value is -1, indicating an array-related object, then the object will be allocated from the protected heap; Otherwise, this variable is the basic unit size of an object. Prober further collects the actual size for the allocation. When the allocation size is multiple times of the basic structure, then Prober decided that the current allocation is an array-related object. After that, the object will be allocated from the protected heap. Note that when there is an extensive number of array-related allocations, Prober allows users to disable the protection on some allocation sites explicitly via a blacklist file. In particular, Prober-Static generates a unique ID for each allocation site, then users can specify the IDs of allocation sites that should be excluded for the protection.

For each deallocation, Prober should determine whether this object is coming from the protected heap. Prober utilizes the address of the deallocation to determine it, where the address must be located in a special range. For such objects, Prober returns them to the protected heap as described in Section 3.2.2. Otherwise, objects will be passed to the default allocator.

3.2.2 Management of Protected Heap. Prober designs its own heap to manage array-related objects in order to reduce the performance overhead. Similar to existing allocators [3, 16, 31], Prober manages small and large objects separately. The idea behind this is that large objects are typically much less, and then they do not have a big chance of being re-utilized. Objects larger than 31 pages are treated as large objects, which are allocated from or returned to the OS directly by invoking `mmap` and `munmp` system calls. In order to determine whether an object is a protected big object, Prober maintains a hash table to track addresses of susceptible large objects, and confirms it by checking the hash table. The protected big object will be returned to the OS with the `munmap` system call.

Objects less than 31 pages will be treated as small objects, which are managed differently from big objects. Prober overcomes multiple design issues of Electric Fence. First, Electric Fence introduces high contention for multi-threaded applications with one global array to hold all freed objects. For instance, every freed object can be stored into a global array only after holding the global lock, preventing concurrent allocations and deallocations from multiple threads. Second, it uses the best-fit allocation policy. For each allocation, it searches the whole array to find the best-matched buffer, which is unnecessarily slow. If it fails to find one, it either divides a bigger object into two parts, or maps a new one from the OS directly. Third, it supports the coalescence and splitting of objects, which invokes unnecessary `mprotect` system calls to change the attributes of protection. Instead, Prober takes the opposite approaches of Electric Fence to improve the performance.

Fixed Size Class: The size of each object is kept the same during the whole execution. Therefore, there is no need to invoke `mprotect` to change the protection attribute. It is intuitive to maintain 31 classes, starting from 1-page to 31-page, but this method does not encourage memory utilization. Instead, Prober maintains only five size classes, including 1-page, 3-pages, 7-pages, 15-pages, and 31-pages. All of these size classes are one page less than power-of-two pages, since one page is reserved for the protected page. Given this design, Prober could quickly compute the bag index using simple bit-shifts operations, which is integrated with its next information-computable design.

Information-Computable Design: Upon every deallocation, Prober checks the size of each object in order to return it to the freelist belonging to the corresponding size class. One naive design is to maintain the size of each object into a hash table, which may invoke extensive searching and comparing operations. Instead, Prober adopts the "information-computable design" of existing work [41, 42], as shown in Figure 3. It takes advantage of the vast virtual address space of 64-bits machines. Prober maps a large chunk of virtual memory from the underlying operating systems at first, and then divides it into multiple regions (called as "bags") with the same size. Each bag only holds objects with the same size class. This design enables the quick computation of the bag index (thus the size of a given object) by the address, which can be computed by dividing the offset with the size of each bag.

Per-Thread Heap: In order to reduce lock contention of multi-threaded applications, Prober adopts the per-thread heap idea of Hoard [3]: allocations and deallocations of different threads will occur only in their own per-thread heaps, without the acquisition of a global lock. Only when freed objects of a per-thread heap are larger than a predefined threshold, then these freed objects will be returned to the global buffer and then be shared by all other threads. In order to quickly locate the index of a thread, Prober intercepts the creation of threads in order to assign a thread index for every thread, which will be stored in its Thread Local Storage (TLS). Therefore, upon each allocation and deallocation, Prober could quickly locate its per-thread heap using its thread index, and then direct it to its per-thread heap.

Pre-allocated FreeArray: Prober utilizes a pre-allocated circular array to track available/freed objects. Two continuous deallocations will be stored next to each other (except the last one in the array). For allocations from the FreeArray, Prober utilizes the Last-In-First-Out (LIFO) algorithm, since the most-recently-freed object have a larger chance of being in the cache, which improves the cache efficiency. Comparing to the normal freelist that each entry will be getting from a new allocation, the array-based design improves the cache efficiency, since multiple continuous allocations and deallocations can be satisfied from objects stored in the same cache line. Its pre-allocated array further avoids the overhead of allocations of free lists.

Overall, Prober manages memory as follows. Upon each memory allocation, Prober determines the size class by rounding the allocation size up to its next size class. After that, the FreeArray of its specific size class will be checked first. If a freed object is available, the request will be satisfied with the FreeArray. When there is no freed objects, Prober fetches multiple objects together from the never-allocated ones to the FreeArray, and then allocates one from it. During each deallocation, Prober first checks whether the deallocation is allocated from the protected heap or not. If an object is not from the protected heap, Prober invokes the default allocator to deal with that. Otherwise, the current freed object will be added into the per-thread FreeArray with the corresponding size class. Note that the allocator only saves the starting address of the freed block to the FreeArray. If the FreeArray is full, half of the freed objects will be donated to the global buffer. That is, Prober only involves with lock operations when there are no freed objects in the per-thread FreeArray, or when the per-thread FreeArray is full. Therefore, Prober's design minimizes the lock contention.

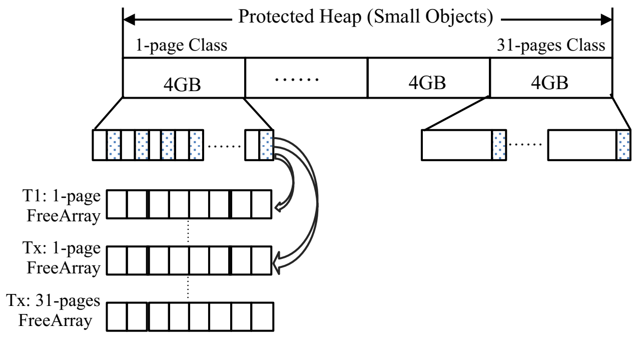

Figure 3: Basic idea of heap design.

3.2.3 Exception Handler. Since both overflows and program failures could trigger segmentation faults, Prober should determine whether a fault is caused by an overflow or not, and only report the faulty instruction to users for heap overflows. To achieve this target, Prober registers its segmentation fault handler in order to capture every *SIGSEGV* signal. Inside the signal handler, Prober first checks whether the current object is one of the protected objects, either the small or the big one. If the access does not belong to the protected object, the exception will be passed to the default handler of applications. Otherwise, the current fault is an overflow. Prober reports the faulty instruction differently, depending on whether the binary includes the symbol information. If the symbol information is included, Prober reports the detailed line number information with the `addr2line` command. Otherwise, Prober only reports binary addresses of the corresponding call stacks. For each overflow, Prober also reports the callstack of its corresponding memory allocation site, where the information will be stored in the shadow memory for small objects and in the hash table for big objects.

4 EXPERIMENTAL EVALUATION

We performed the experiments on a two-socket quiescent machine, where each socket is an Intel(R) Xeon(R) Gold 6138 processor with 20 cores. It has 200GB main memory, and 32KB L1, 1024 KB L2 and 28160 KB L3 cache. The experiments were performed on Ubuntu 18.04, installed with Linux-4.15.0 kernel. All applications were compiled with LLVM-8.0, by adding an analysis/instrumentation pass of Prober-Static.

4.1 Effectiveness

The effectiveness evaluation includes two parts, 38 bugs included in the existing study [47] and other overflow bugs included in other existing work, such as Bugbench [28], CVE database, or HeapTherapy [49].

4.1.1 38 Bugs from the Existing Study. For 38 bugs listed in the existing study, we confirm that Prober correctly instrumented 36 bugs out of them. The remaining two bugs cannot be instrumented due to the invocation of external standard library calls (e.g., libstdc++), which are not analyzed (shared by instrumentation-based approaches). Therefore, Prober's evaluation presents high confidence on the actual overhead, since it could instrument all bugs correctly.

Note that we did not run these buggy applications directly, due to the following reasons. First, these bugs may not include erroneous inputs that are required to exercise them. Second, many of them are not compatible with modern libraries, which requires a significant amount of manual efforts for the compilation. Therefore, we only verify whether the corresponding bugs have been instrumented correctly.

4.1.2 Other Real-world Bugs. We performed the effectiveness evaluation on the other real-world 10 bugs that are not listed in the existing study [47]. These applications and their specific bug trigger inputs are obtained from Bugbench [28] , CVE database, or HeapTherapy [49]. Among these 10 vulnerable applications, the heartbleed and libtiff-4.0.7 vulnerabilities are caused by buffer over-reads, while others are caused by buffer over-writes. The details of these applications are shown in Table 5, where all of these bugs can be detected by AddressSanitizer. Table 5 also listed the number of allocation sites that can be identified statically ("Static" column) and dynamically ("Dynamic"). Overall, Prober detects all known overflows without false positives. Upon detection, Prober stops the execution immediately (before the crashes), and reports the type of an overflow (over-read or over-write), the call path of triggering the overflow, and the allocation site of the corresponding buffer. The evaluation confirms that Prober is able to detect real heap overflows with its proposed instrumentation and runtime system.

4.1.3 Case Study. Figure 4 shows the bug report for the heartbleed vulnerability. Prober identifies that this bug is a buffer over-read problem. The bug report also includes the call stack of the faulty instruction (where the overflow occurs), and the call stack of this object's allocation site.

According to the bug report, the overflow occurs in the *memcpy*() function, which is invoked by the *tls1_process_heartbeat* function at line 2586 of `./ssl/t1_lib.c` file. By checking the source code, the corresponding statement is $memcpy(bp, pl, payload)$. According

Table 5: Statically and dynamically identified callsites in buggy applications

Application	Reference	Static (#)	Dynamic (#)
bc-1.06	BugBench [28]	43	5
gzip-1.2.4	BugBench [28]	3	1
Heartbleed	CVE-2014-0160 [13]	9314	3941
LibHX-3.4	CVE-2010-2947 [6]	23	15
Libtiff-4.0.1	CVE-2013-4243 [7]	406	75
Libtiff-4.0.7	CVE-2016-10269 [9]	421	104
Memcached-1.4.25	CVE-2016-8706 [45]	80	19
openjpeg-1.3	CVE-2012-3535 [39]	756	201
polymorph-0.4.0	BugBench [28]	1	0
squid-2.3	BugBench [28]	83	175

```
A buffer over−read problem is detected at:
../glibc/../multiarch/memcpy−avx−unaligned.S:237
../x86_64−linux−gnu/bits/string3.h:53
../openssl−OpenSSL_1_0_1f/ssl/t1_lib.c:2586
../openssl−OpenSSL_1_0_1f/ssl/s3_pkt.c:1092
../openssl−OpenSSL_1_0_1f/ssl/s3_both.c:457
...
../nginx−1.3.9/src/event/ngx_event.c:247
../nginx−1.3.9/src/os/unix/ngx_process_cycle.c:807
...
This object is allocated at:
../openssl−OpenSSL_1_0_1f/ssl/s3_both.c:770
../openssl−OpenSSL_1_0_1f/ssl/s3_pkt.c:949
../openssl−OpenSSL_1_0_1f/ssl/s3_both.c:457
```

Figure 4: Bug report for the Heartbleed Problem.

to the attribute of this problem–a buffer over-read problem, it is easy to know that the over-read issue is related to the source of `memcpy`, that is, either *pl* or `payload`. Since *pl* is the starting address of a normal heap object that is allocated at line 770 of `./ssl/s3_both.c` file, then the failure must be caused by `payload`. By examining the source code, we could easily find out that `payload` is computed from the length of the data that the server receives from the network. Therefore, via the bug report, programmers can easily reason the root cause of overflow, and fix the problem correspondingly.

4.2 Performance Overhead

To evaluate the performance overhead, Prober is evaluated on a popular benchmark suite–PARSEC [4], and multiple widely-utilized real applications, such as sqlite, memcached, aget, pbzip2, and pfscan, with 18 multithreaded applications in total. For PARSEC benchmarks, we used the native inputs and 40 threads. For applications that can only use power-of-2 threads, e.g., facesim, then they will use 32 threads [4].

We compare Prober with Prober-All, Electric Fence [37] ,and AddressSanitizer [40]. Prober-All protects all heap objects through the page protection despite they are array-related or not. Electric Fence also utilizes the page protection to protect all heap objects, which is very similar to Prober-All, but with different implementation. AddressSanitizer is an instrumentation-based approach that checks every memory access, representing one important technique that is widely employed in development phases. For both AddressSanitizer and Prober, we did not instrument any external and standard libraries that are required by these applications, which could impose more overhead if they are included. For the fair comparison, we disable the checks of global and stack overflows for AddressSanitizer.

Figure 5 shows the normalized runtime of these four systems, which are normalized to the runtime of the default Linux libraries. Overall, the average overhead of Prober, Prober-All, Electric Fence, and AddressSanitizer are 1.5%, 2.4×, > 7×, 42.9%, respectively. The largest overhead of Prober is only 9.3% for `ferret`.

For both AddressSanitizer and Electric Fence, `freqmine` is crashed due to an unknown problem in our evaluation environment. `aget` is crashed with Electric Fence as well. For Electric fence, five applications, including `canneal`, `dedup`, `facesim`, `raytrace` and `swaptions`, cannot finish the execution within 1 hour (marked as "T"). For Prober, dedup runs over 2× slower initially, if all identified callsites are protected. Based on our analysis, one callsite has over 50% of allocations, which is excluded manually (as described in Section 3.2.1).

Multiple reasons contribute to the big performance difference of these systems. AddressSanitizer's performance overhead mainly comes from its checking overhead for every memory access, which also explains its little overhead for IO-bound applications, such as `aget` or `pfscan`. Comparing to AddressSanitizer, Prober does not check on accesses, with its page protection mechanism. Comparing to Prober-All and Electric Fence, Prober only protects array-related objects, instead of all heap objects. Therefore, the overhead of Prober-All and Electric Fence is much higher than that of Prober. Electric Fence imposes the highest overhead due to its implementation issues as discussed in Section 3.2.2.

Comparing to Electric Fence, Prober-All (and Prober) is more efficient due to multiple reasons. First, Electric Fence utilizes the global lock to protect all allocations and deallocations, which unfortunately serializes all allocations and deallocations. Second, Electric Fence may coalesce continuous objects upon deallocations, and split a larger object to smaller ones upon allocations. Third, Electric Fence cannot quickly locate the metadata information. Instead, Prober and Prober-All designs per-thread heap, pre-allocated freeArray, and information-computable design as further discussed in Section 3.2.2.

We further collected the characteristics of these applications dynamically, as shown in Table 6. In this table, the "`Total Objects`" column shows the total number of allocations for each application, including allocations from the application and all libraries. The "`Protected Objects`" column shows the number of allocations that are protected in total. Here, "`Static`" and "`Dynamic`" represent whether such objects can be identified statically or dynamically (requiring the determination of its runtime). Among these protected objects, "Live" column shows the maximum number of objects that are protected at the same time. The "`Unprotected Objects`" column indicates the number of heap objects that are not protected by Prober. We have the following observations.

First, most applications have a larger portion of objects that are not array-related, such as `canneal`, `raytrace`, and `vips`. This

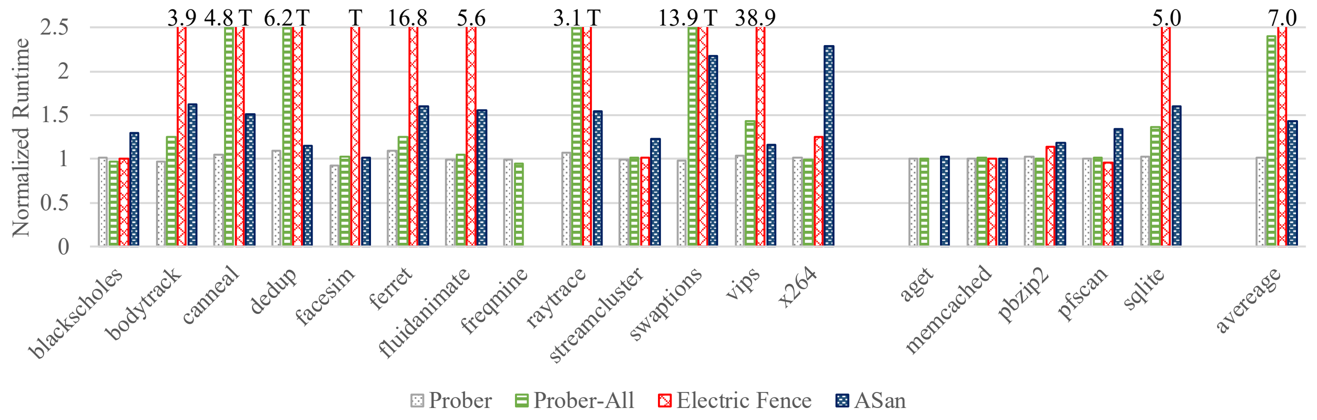

Figure 5: Performance overhead of Prober, Prober-All, Electric Fence and AddressSanitizer, where they are all normalized to the runtime of the default Linux libraries. Applications with the bar marked as "T" (indicating the timeout) cannot finish the execution in one hour.

Table 6: Characteristics of applications.

Application	Total Objects	Protected Objects			Unprotected Objects
		Static	Dynamic	Live	
blackscholes	208	5	0	5	203
bodytrack	452,084	2,349	0	70	449,735
canneal	21,141,661	0	0	0	21,141,661
dedup	1,887,373	734	336,747	140,630	1,549,892
facesim	4,908,999	2,302,579	432	440,944	2,605,988
ferret	549,109	380,546	3,496	81,736	165,067
fluidanimate	230,105	24	0	24	230,081
freqmine	8,699	7,555	0	944	1,144
raytrace	20,000,619	12	1	12	20,000,606
streamcluster	9,241	8,854	6	15	381
swaptions	48,001,995	48,001,796	0	1,826	199
vips	1,430,261	0	1,312	1,312	1,428,949
x264	37,332	46	37,121	2,355	165
aget	319	38	0	29	281
memcached	848	276	33	309	539
pbzip2	23,851	0	0	0	23,851
pfscan	238	3	1	3	234
sqlite	2,889,043	2,888,868	0	5,826	175

indicates that our key insight is effective in reducing the scope of protection.

Second, in Prober, the number of objects that are protected at the same time (in "`Live`" column) will affect the performance overhead, but not the number of protected objects, since freed objects are re-used in Prober. The overhead of each protected object comes from two aspects. First, it comes from the `mprotect` system call to insert the protected page. After that, page protection imposes no additional overhead for checking the overflow. Second, a large number of protected objects (in different pages) may increase page faults. This explains why dedup, `ferret`, and `sqlite` impose higher performance overhead than others. However, `facesim` imposes a low overhead, although with the largest number of live objects. This is due to the fact that `facesim` runs much longer than other applications, where the averaged number of protection is still small. In addition to that, `facesim` has found to have serious memory leaks [20], indicating that the number of live objects is smaller than that in Table 6. This indicates that `facesim` may not increase the number of page faults.

Third, Table 6 explains that some applications require a long execution with Prober-All, Electric Fence, such as `canneal`, `raytrace`, and `swaptions`, since there are a large number of objects inside. Prober avoids this issue by protecting only array-related objects, instead of all objects. Table 6 also explains why Prober-All and Electric Fence performs well in `blackscholes`, `streamcluster`, `memcached`, `pbzip2`, and `pfscan`, since only a few allocations exist in these applications. Prober's unique observation and its efficient heap design (as discussed in Section 3.2.2) make it efficient enough for the deployment environment, but without compromising its effectiveness.

4.3 Memory Overhead

We also evaluated the memory overhead of Prober using the same applications that are used in the performance evaluation. To collect memory consumption of server applications, such as `memcached`, a script is designed to periodically collect the `/proc/PID/status` file. Then the maximum value of the `VmHWM` field is utilized as the maximum memory consumption. For other applications, memory consumption is collected from the output of the `time` utility, where the `maxresident` field reports the maximum memory consumption of an application [2].

The real memory data is omitted due to space limitations. In total, Prober utilizes 25.9% more memory when compared to the default library. In contrast, Electric Fence utilizes around 3.9× memory, and AddressSanitizer's memory overhead is around 69.7%. That is, Prober utilizes significantly less memory than Electric Fence and AddressSanitizer. We also observed that applications with a small footprint have a higher memory overhead, coming from the storing of thread information, heap information, and other metadata that are not proportional to their memory usage.

5 LIMITATIONS

Prober focuses on array-related heap overflows, representing over 86% of heap overflows based on our observations (Section 2). It cannot detect array-related internal-structure overflows, which is its biggest limitation. However, there is no fundamental reason why this cannot be done. It is possible to arrange the fields of the structure so that array(s) can be placed at the end of the corresponding structure. Adding the support for internal-structure overflows will be our future work.

Prober can only detect overflows landing on the protection page(s). Prober can be configured to change the pages for the protection if necessary. In theory, it is able to detect more errors than existing approaches with redzones, such as AddressSanitizer [40]. It currently cannot detect heap underflows. However, heap underflows cannot do any harm, since they can only land on the non-used area.

Prober only detects overflows when the source code is analyzed and instrumented by Prober-Static. This limitation is also shared by all instrumentation-based tools, e.g., EffectiveSan [11] or AddressSanitizer [40]. When an overflowing object is allocated in a library that is not instrumented, Prober cannot detect it. However, different from existing work that detects overflows by checking memory accesses, Prober can detect overflows caused by APIs of a non-instrumented library. This is a significant difference.

6 RELATED WORK

We classify existing tools of detecting heap buffer overflow based on the type of approaches.

Static Detection: Many tools utilize static analysis to detect buffer overflow bugs [14, 22, 23, 27]. They only analyze software source code in order to reason which statements could potentially cause buffer overflows. However, some variables (e.g., indirect branches) could not be determined without the execution. Thus, they may generate many false positives or false negatives, which requires further manual efforts to confirm the reported bugs. In contrast, Prober never generates any false positives.

Dynamic detection: Several tools place an inaccessible memory page around every heap object [32, 33, 37, 49], which is similar to Prober. Memory accesses to the protected pages will generate a SIGSEGV signal. However, these existing work suffer from a prohibitively high performance overhead by protecting all pages or even probabilistically. Although Prober employs the same mechanism to detect heap buffer overflow, it narrows down heap objects that can potentially result in buffer overflows, which drastically reduces its performance overhead. Also, Prober designs its runtime system carefully to reduce its overhead.

Static instrumentation-assisted detection: Numerous tools analyze source code to identify necessary instrumentation, which favors sanity checks at runtime [1, 8, 12, 15, 17, 21, 29, 36, 38, 40]. They instrument all memory accesses at compilation phases, and check the validity of accesses at runtime. AddressSantizer [40] is the state-of-art of this type of approaches, which further employs the static analysis to prune out certain unnecessary checks. However, AddressSanitizer still imposes non-negligible performance overhead, as further evaluated in Section 4.2. Different from these tools, Prober does not check every memory access, but relying on the page protection to detect overflows without checking overhead, if there is no overflow.

Dynamic instrumentation-assisted detection: A lot of dynamic analysis tools detect memory errors based on the checking of memory accesses during runtime, such as Valgrind's Memcheck tool [30], Dr. Memory [5], Purify [18], Intel Inspector [19], and Sun Discover [35]. Due to the expensive instrumentation and inspection, they typically impose too high-performance overhead to be employed in the production environment.

Hardware-assisted detection: A few tools rely on new hardware to detect buffer overflows. Intel MPX tries to reduce the overhead of pointer checks by embedding checks into a new hardware [34]. BOGO relies on Intel MPX to provide both spatial and temporal safety [50]. However, the overhead of validating every memory access is too high to be adopted in practice. Sampling-based techniques, such as CSOD [25] and Sampler [43], utilize hardware watchpoints or Performance Monitor Unit (PMU) hardware to monitor a few heap objects at one time or validate a subset of memory accesses. Although they impose low runtime overhead similarly as Prober, they cannot guarantee the same effectiveness as Prober, especially when there are a lot of heap objects. CHERI requires the cooperation of architecture, compiler, and operating system together to enforce memory safety [46], which inevitably increases developers' effort. Prober, which is a dynamically linked library, imposes little manual effort, without changing the underlying OS and requiring new hardware.

Postmortem detection: Some evidence-based tools detect buffer over-writes by appending canaries after each heap object and checks if canaries are corrupted at memory deallocations or epoch ends [24, 26, 48, 49]. Since read operations do not leave evidence, they cannot detect read-based buffer overflow, while Prober can detect both buffer over-reads and buffer over-writes. Also, evidence-based approaches cannot be applied in the security environment, since the attacks may already be issued successfully before performing the detection.

7 CONCLUSION

This paper presents a novel system to defend heap overflows. It is based on a key observation that is obtained from the analysis of 48 real overflow bugs: overflowing objects are typically involved with arrays. Based on this observation, Prober takes a two-phase approach to detect heap overflows: its static component identifies all possible array-related allocations before the compilation, and then instruments the code correspondingly; Its dynamic component further intercepts the allocations, and redirects the allocations from susceptible allocation sites to the protected heap in order to detect the overflows with the page protection mechanism. Overall, Prober only imposes around 1.5% performance overhead on average, but without compromising its effectiveness. The low overhead and the high effectiveness makes Prober an always-on approach for the production environment.

8 ACKNOWLEDGEMENTS

We would like to thank all anonymous reviewers for their constructive suggestions and feedback. This paper is partially supported by the National Science Foundation under Award CCF-2024253, UMass Amherst startup package, and William & Mary startup package.

REFERENCES

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th conference on USENIX security symposium* (Montreal, Canada) *(SSYM'09)*. USENIX Association, Berkeley, CA, USA, 51–66. http://dl.acm.org/citation.cfm?id=1855768.1855772

[2] Andries Brouwer. 2015. *time - time a simple command or give resource usage.* Linux Comunity.

[3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. 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)*. Association for Computing Machinery, New York, NY, United States, Cambridge, MA, 117–128. citeseer.ist.psu.edu/berger00hoard.html

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. Association for Computing Machinery, New York, NY, United States, 1–10.

[5] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 213–223. http://dl.acm.org/citation.cfm?id=2190025.2190067

[6] Bugzilla. 2010. "libHX: buffer overrun in HX_split()". https://bugzilla.redhat.com/show_bug.cgi?id=625866.

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

[8] Zhe Chen, Junqi Yan, Shuanglong Kan, Ju Qian, and Jingling Xue. 2019. Detecting Memory Errors at Runtime with Source-level Instrumentation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 341–351. https://doi.org/10.1145/3293882.3330581

[9] The MITRE Corporation. 2016. CVE-2016-10269. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10269.

[10] CVEdetails. 2019. Vulnerabilities By Type. https://www.cvedetails.com/vulnerabilities-by-types.php.

[11] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 181–195. https://doi.org/10.1145/3192366.3192388

[12] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 181–195. https://doi.org/10.1145/3192366.3192388

[13] Exploit. 2014. "Openssl heartbeat poc with starttls support". https://gist.github.com/takeshixx/10107280.

[14] Micro Focus. 2019. Fortify Static Code Analyzer. https://www.ndm.net/sast/hp-fortify. last visited: 02/08/2019.

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

[16] Sanjay Ghemawat and Paul Menage. 2005. TCMalloc : Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[17] Niranjan Hasabnis, Ashish Misra, and R. Sekar. 2012. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12)*. ACM, New York, NY, USA, 135–144. https://doi.org/10.1145/2259016.2259034

[18] Reed Hastings and Bob Joyce. 1992. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*. USENIX Association, Berkeley, Califonia, USA, 125–138.

[19] Intel Corporation. 2012. Intel Inspector XE 2013. http://software.intel.com/en-us/intel-inspector-xe.

[20] Changhee Jung, Sangho Lee, Easwaran Raman, and Santosh Pande. 2014. Automated Memory Leak Detection for Production Use. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 825âĂŞ836. https://doi.org/10.1145/2568225.2568311

[21] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta Pointers: Buffer Overflow Checks Without the Checks. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. ACM, New York, NY, USA, Article 22, 14 pages. https://doi.org/10.1145/3190508.3190553

[22] David Larochelle and David Evans. 2001. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10* (Washington, D.C.) *(SSYM'01)*. USENIX Association, Berkeley, CA, USA, Article 14, 177âĂŞ190 pages. http://dl.acm.org/citation.cfm?id=1251327.1251341

[23] Wei Le and Mary Lou Soffa. 2008. Marple: A Demand-driven Path-sensitive Buffer Overflow Detector. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 272–282. https://doi.org/10.1145/1453101.1453137

[24] Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian, and Tongping Liu. 2018. 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* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 344–358. https://doi.org/10.1145/3192366.3192380

[25] Hongyu Liu, Sam Silvestro, Xiaoyin Wang, Lide Duan, and Tongping Liu. 2019. CSOD: Context-Sensitive Overflow Detection. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 50âĂŞ60.

[26] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: Fast and Precise Error Detection via Evidence-based Dynamic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 911–922. https://doi.org/10.1145/2884781.2884784

[27] Checkmarx Ltd. 2019. Checkmarx. https://www.checkmarx.com. last visited: 02/08/2019.

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

[29] George C. Necula Necula, McPeak Scott, and Weimer Westley. 2002. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the Principles of Programming Languages*. Association for Computing Machinery, New York, NY, United States, 128–139.

[30] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (San Diego, California, USA) *(PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[31] Gene Novark and Emery D. Berger. 2010. DieHarder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security* (Chicago, Illinois, USA) *(CCS '10)*. ACM, New York, NY, USA, 573–584. https://doi.org/10.1145/1866307.1866371

[32] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2007. Exterminator: automatically correcting memory errors with high probability. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007)* (San Diego, California, USA). ACM Press, New York, NY, USA, 1–11. https://doi.org/10.1145/1250734.1250736

[33] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2009. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2009)* (Dublin, Ireland). ACM, New York, NY, USA, 397–407. https://doi.org/10.1145/1542476.1542521

[34] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 2, Article 28 (June 2018), 30 pages. https://doi.org/10.1145/3224423

[35] Oracle Corporation. 2011. Sun Memory Error Discovery Tool (Discover). http://docs.oracle.com/cd/E18659_01/html/821-1784/gentextid-302.html.

[36] parasoft Company. 2013. *C and C++ Memory Debugging.*

[37] Bruce Perens. 2005. Electric Fence. https://linux.softpedia.com/get/Programming/Debuggers/Electric-Fence-3305.shtml.

[38] Olatunji Ruwase and Monica S. Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*. The Internet Society, San Diego, California, USA, 159–169.

[39] Kurt Seifried. 2012. "CVE Request: Heap-based buffer overflow in openjpeg". https://seclists.org/oss-sec/2012/q3/300.

[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 28–28. http://dl.acm.org/citation.cfm?id=2342821.2342849

[41] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. 2017. FreeGuard: A Faster Secure Heap Allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. ACM, New York, NY, USA, 2389–2403. https://doi.org/10.1145/3133956.3133957

[42] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. 2018. Guarder: A Tunable Secure Allocator. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 117–133. https://www.usenix.org/conference/usenixsecurity18/presentation/silvestro

[43] Sam Silvestro, Hongyu Liu, Tong Zhang, Changhee Jung, Dongyoon Lee, and Tongping Liu. 2018. Sampler: PMU-Based Sampling to Detect Memory Errors

Latent in Production Software. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 231–244.

[44] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, USA, 48–62. https://doi.org/10.1109/SP.2013.13

[45] Talos. 2016. "Memcached Server SASL Autentication Remote Code Execution Vulnerability". https://www.talosintelligence.com/reports/TALOS-2016-0221/.

[46] 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. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. 20–37. https://doi.org/10.1109/SP.2015.9

[47] T. Ye, L. Zhang, L. Wang, and X. Li. 2016. An Empirical Study on Detecting and Fixing Buffer Overflow Bugs. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 91–101. https://doi.org/10.1109/ICST.2016.21

[48] Qiang Zeng, Dinghao Wu, and Peng Liu. 2011. 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* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 367–377. https://doi.org/10.1145/1993498.1993541

[49] Qiang Zeng, Mingyi Zhao, and Peng Liu. 2015. 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)*. IEEE Computer Society, Washington, DC, USA, 485–496. https://doi.org/10.1109/DSN.2015.54

[50] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2019. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 631–644. https://doi.org/10.1145/3297858.3304017