CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++

Jangseop Shin*, Donghyun Kwon*, Jiwon Seo*, Yeongpil Cho†‡ and Yunheung Paek*
*ECE and ISRC, Seoul National University
{jsshin, dhkwon, jwseo}@sor.snu.ac.kr, ypaek@snu.ac.kr
†School of Software, Soongsil University
ypcho@ssu.ac.kr

***Abstract*—Pointer invalidation has been a popular approach adopted in many recent studies to mitigate use-after-free errors. The approach can be divided largely into two different schemes: explicit invalidation and implicit invalidation. The former aims to eradicate the root cause of use-after-free errors by explicitly invalidating every dangling pointer. In contrast, the latter aims to prevent dangling pointers by freeing an object only if there is no pointer referring to it. A downside of the explicit scheme is that it is expensive, as it demands high-cost algorithms or a large amount of space to maintain up-to-date lists of pointer locations linking to each object. Implicit invalidation is more efficient in that even without any explicit effort, it can eliminate dangling pointers by leaving objects undeleted until all the links between the objects and their referring pointers vanish by themselves during program execution. However, such an argument only holds if the scheme knows exactly when each link is created and deleted. Reference counting is a traditional method to determine the existence of reference links between objects and pointers. Unfortunately, impeccable reference counting for legacy C/C++ code is very difficult and expensive to achieve in practice, mainly because of the type unsafe operations in the code. In this paper, we present a solution, called *CRCount*, to the use-after-free problem in legacy C/C++. For effective and efficient problem solving, CRCount is armed with the *pointer footprinting* technique that enables us to compute, with high accuracy, the reference count of every object referred to by the pointers in the legacy code. Our experiments demonstrate that CRCount mitigates the use-after-free errors with a lower performance-wise and space-wise overhead than the existing pointer invalidation solutions.**

I. Introduction

Use-after-free (UAF) errors refer to unlawful dereferences of *dangling pointers*, which are the pointers that still point to a freed and thus stale object. UAF errors constitute a serious threat to software security because they are considered significantly difficult to identify by compilers and manual analyses. This difficulty is attributed to the fact that the temporal distances between arbitrary pointer operations, such as setting a pointer to the address of the object, freeing the object, and dereferencing the pointer, can be very long and hence very difficult to analyze accurately in reality. This difficulty has led attackers to leverage UAF errors as a primary source for exploitation in their attempts [40], [20], [38] to access or corrupt arbitrary memory locations in a victim process.

In the past decade, mitigation against UAF errors has been approached by many researchers from various directions. In one group of studies [33], [24], [21], [22], [31], researchers attempted to detect the UAF error when a pointer is dereferenced to access its referred object (or *referent*). Their goal is to validate the access to the object by carrying out a sequence of operations to check whether the referent is stale. To support this *access validation* mechanism, each time an object is allocated, they label the object with a unique *attribute* that identifies the allocation. Later, when a pointer is dereferenced, they examine the attribute of its referent to check whether or not the access is made by a dangling pointer whose referent no longer holds the original valid allocation in memory.

Although mitigation techniques based on access validation are claimed to be extensive and complete, they tend to incur an excessively high performance overhead. This high overhead is attributed to the fact that the attribute checks must be executed exhaustively for every memory access, thereby considerably increasing the total execution time. More recently in a different group of studies, as a new direction of UAF defense research to reduce this performance overhead, some researchers have proposed an approach based on *pointer invalidation* [17], [40], [36]. Their mitigation approach against UAF errors is to deter the violations preemptively by getting rid of the dangling pointers at the outset. As a pointer becomes dangling when its referents get freed, this approach in principle can succeed by invalidating all the related pointers when an object is freed such that an exception is triggered when one of the invalidated pointers is dereferenced afterwards. However in practice, for this approach to be successful, we need to address the problem of accurately tracking down every change, such as the creation, copy, or destruction of pointers, and hence, of identifying pointers and their referents located anywhere on the execution path. Unfortunately, this *pointer tracking* problem in general is prohibitively difficult and expensive to solve with high accuracy because the pointers may be copied into a number of different data structures during program execution.

For precise pointer tracking, DANGNULL [17] uses dynamic range analysis to monitor the pointer arithmetic operations that alter the reference relationships between the pointers and the memory objects. Unfortunately, DANGNULL suffers from a high performance overhead. A majority of this overhead is attributed to the design element that requires the system

‡Corresponding author.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23541
www.ndss-symposium.org

to immediately update the metadata for the objects when there is a change in the reference relationships. To alleviate this performance overhead, DangSan [36] takes a different approach wherein the total cost for updating the reference relationships is reduced by discarding the transitional changes intermittently produced in a sequence of pointer arithmetic operations. More specifically, in this approach, when any of the existing reference relationships is changed by pointer arithmetic, this change is not reflected immediately in the relationships (thus saving CPU cycles); instead, the change is merely stored in a history table as a record for future reference. The actual reference relationships are checked later when the object is freed. Experiments on DangSan have proven the effectiveness of this approach by showing that it achieved a considerably lower performance overhead than DANGNULL. However, the experiments also show that the history table can become unbearably large when benchmark programs use pointers intensively. For example, the memory overhead of `omnetpp` benchmark was more than a hundred times the original memory consumption. As UAF errors are more likely to be prevalent in programs with a heavy use of pointers, such an immense memory overhead might be a significant obstacle for a broad application of this approach.

From the observations on previous work, we found that such a high overhead, either performance-wise or space-wise, of the existing pointer invalidation techniques is basically caused by the approach that when an object is freed, these techniques promptly locate and explicitly invalidate all the pointers referring to the object. This *explicit pointer invalidation* approach seems to be intuitive as it mitigates UAF errors by eradicating the root cause (i.e. dangling pointers), but it is usually very costly as it demands expensive algorithms or a large amount of space to maintain the up-to-date list of pointer locations linking to each object at all times during program execution. DANGNULL spends many CPU cycles to manage binary trees as the data structures to store pointer locations. Every time there is a change in one of the locations, the trees are traversed and modified accordingly, consuming a considerable amount of the execution time. Even worse, the total performance overhead increases in proportion to the numbers of pointers and referents, which can increase considerably for programs, such as `omnetpp`, that perform frequent arithmetic operations on a myriad of pointers.

Our findings motivated us to take an alternative approach, which we have named *implicit pointer invalidation* to contrast with the existing explicit approach. The goal of our approach is to prevent dangling pointers by enforcing a basic principle that permits an object to be freed only if there is no pointer currently referring to it. Of course in C/C++, users may deallocate an object at their disposal by invoking the `free` (or `delete`) function irrespective of the existence of pointers linking with the object. Therefore, to enforce the principle in the legacy C/C++ code, we augment each memory object with a single integer, known as the *reference counter*, that records the number of pointers referring to the designated object. When the user intends to `free` an object in the original code, we ignore the function by doing nothing explicit if the corresponding reference counter has a non-zero value. The object is disposed of without explicit effort for invalidation once the counter comes to zero. Indeed, in most real code, reference counters eventually decrease to zero in a sequence of repeated pointer operations, such as assignment, nullification, and deallocation of pointer variables. This implies that even without explicit invalidation time and effort, the proposed scheme can prevent dangling pointers by holding an object remain undeleted until all the links between the object and its referring pointers vanish by themselves, which is tantamount to the implicit invalidation of the referring pointers.

This implicit invalidation scheme sounds plain and naive at the first glance, but its practical application to the existing C/C++ code is very challenging from several aspects. The first aspect of concern is the increase in the memory overhead. In C/C++, `free`/`delete` is purposed to instantly release the memory space occupied by objects and reclaim the space for reuse. However, such reclamation of memory will be hindered by our implicit scheme that delays the release of a to-be-free object, which thus remains *undeleted* until its reference count reduces to zero. Consequently, our scheme could suffer from a memory overhead due to undeleted (and thus unreclaimed) objects, particularly if their number becomes large. Luckily, as will be empirically demonstrated, the overhead was manageably small for most real cases as far as we could accurately compute the reference counts and timely delete the undeleted objects. In fact, this very problem of reference count computing is another important aspect to be considered for the practical application of our scheme to legacy code because the notorious C/C++ programming practices heedlessly violating type safety tend to extremely complicate this problem. For example, common practices, such as unrestricted pointer arithmetic and abusive uses of type casts or unions in C/C++, make it difficult to pinpoint exactly when pointers are generated and deleted at runtime, which in turn results in imprecise and incorrect reference counting.

Because the legacy C/C++ code is such full of type unsafe operations, previous attempts based on reference counting could not effectively tackle the UAF problem in the legacy code [9], [2]. In this paper, we propose *CRCount*, an effective and efficient solution developed to mitigate UAF errors on the basis of implicit invalidation. As reasoned above, the key to the success of our solution depends on the accuracy of reference counting. To compute reference counts with high precision, CRCount adopts a technique called *pointer footprinting*, which tracks down the memory locations of live heap pointers along the execution flow. Our pointer footprinting technique is centered around a special data structure, called the *pointer bitmap*, that represents the up-to-date memory locations where the heap pointers are stored. The bitmap is updated by means of program instrumentation coupled with the runtime library. The empirical results show that, assisted by the footprinting technique, CRCount could track C/C++ pointers with a relatively low overhead and compute the reference counts with high accuracy. CRCount is implemented as a compiler pass in LLVM. Therefore, any C/C++ program can be fortified against attacks exploiting UAF errors merely by compiling its source code with CRCount enabled.

II. Related Work

In this section, we will continue the discussion on CRCount by relating it to previous solutions that also aimed to thwart UAF errors in C/C++ code.

Explicit Pointer Invalidation. We divide the explicit pointer invalidation techniques into two folds depending on the manner in which updates on the reference relationships between pointers and objects are reflected. We deem that DANGNULL [17] and FreeSentry [40] update the reference relationships in an *eager* manner because they always update their metadata for pointers and objects right after the pointer arithmetic operations affecting the relationships. In contrast, we deem that DangSan [36] opt for the *lazy* manner in updating these relationships. This enables DangSan to achieve much better performance, but DangSan's memory overhead is often too large, as the size of the history table grows extremely large for programs with heavy uses of pointers. In principle, CRCount embraces the same eager update strategy as DANGNULL in such a way that when an object is linked/delinked with a pointer by pointer arithmetic, the reference relationships are updated instantly by modifying the object's reference count accordingly. However, CRCount does not suffer from the performance issue as it manages much lighter metadata. Moreover, our implicit pointer invalidation scheme does not suffer from the performance overhead that was mandated by DANGNULL to explicitly invalidate all the pointers referring to an object when the object is freed.

Implicit Pointer Invalidation. Thus far, several studies have come close to CRCount in the sense that they benefit from the implicit pointer invalidation even if this fact is not expressed clearly in the literature [3], [28], [39], [32]. To be more specific, their solutions are exempt from additional force required to explicitly invalidate dangling pointers by delaying the reuse of the recently freed objects in the hope that the number of pointers referring to freed objects would gradually decrease to zero during program execution. However, these approaches differ from CRCount in one important aspect. They do not have notions, such as reference counting, to measure the number of pointer references at runtime. Therefore, they cannot determine exactly how long they should hold the freed objects back from being reused by the memory allocator, and their common schemes are to release the objects simply when specific conditions are met, such as after a random amount of time or when the total size of objects being held reaches a certain limit. Unfortunately, such naive schemes can be easily circumvented by calculated attacks such as heap spraying [17], [8] or heap fengshui [35]. In contrast, CRCount, by maintaining precise reference counters for every object, can guarantee the safe release of freed objects for reuse with no presence of dangling pointers.

Object Access Validation. Many security solutions [24], [33] have attempted to prevent UAF errors by exhaustively validating every object access via pointers. To this end, they use a lock-and-key mechanism that can check the validity by (1) assigning a unique lock to each object at the creation time, and (2) monitoring whether the object accesses are made by the pointers having the correct key matching the target object's lock. This mechanism realizes a thorough defense against UAF errors. However, they are at a disadvantage as compared to CRCount it terms of accuracy and performance: they generate a number of false positive alarms because of their strictness that goes beyond the common programming practices, and incur a huge performance overhead necessary to intervene in every object access.

Secure Layout of Object. Some systems prevent the exploitation of UAF errors by using prudent layouts of objects. Cling [1] forces new objects to be created only in a memory block that has either never been allocated or has been allocated to objects of the same type. In brief, Cling mitigates UAF errors by ensuring the type safety of the allocated objects. Although efficient, it still allows UAF errors between objects of the same type. Oscar [7] defeats UAF errors through a careful arrangement of objects. For this, Oscar never reuses the (virtual) memory, such that all the objects are created in a unique memory space, thereby completely blocking the UAF bugs. Oscar facilitates an effective measure against UAF errors. The downside, however, is that it suffers from a higher performance and memory overhead than CRCount because it abandons the efficiency that could otherwise be gained through the maximal reuse of the memory space.

Garbage Collection. Garbage collection makes a program robust against UAF errors through an automatic mechanism that frees an object after confirming that there is no reference to the object. Unlike the case of JAVA and C# in which garbage collection is built into, no hint to distinguish pointers from ordinary objects is provided by compilers in C/C++. Therefore, Boehm-Demers-Weiser garbage collector (BDW GC) [5], a representative garbage collector for C/C++, uses a conservative approach that regards any pointer-size word as a potential pointer value. Such a conservative approach may result in memory leaks in the case of an erroneous recognition of pointer values, although it has been reported that the problem rarely occurs in 64-bit architectures [14]. Garbage collection is also known to cause a non-negligible memory overhead as it trades space for performance [13]. BDW GC works based on dedicated APIs. Although it provides a way to automatically redirect traditional C memory allocation routines to the corresponding APIs, some porting efforts may be required for large real-world programs, especially for C++ programs.

Smart Pointer. To enforce safe and automatic memory management in C++, an extended data type is provided, called the *smart pointer* [2], which encapsulates a raw pointer with a reference counter. Conceptually, a smart pointer owns one raw pointer, meaning that it is responsible for deleting the object referred to by its raw pointer. During program execution, it keeps track of the reference counter through the language's scoping rules and deletes the referred object from the heap when the reference counter becomes zero. The smart pointer is similar to CRCount in that it is based on the reference counting mechanism. However, there is a critical downside of using smart pointers to enhance memory safety: programmers must take full responsibility of smart pointers. In order for a legacy C++ program to be free from UAF issues, all the raw pointers in the program must be converted manually to smart pointers. Unfortunately, such a complete conversion of every raw pointer is a very time-consuming task to achieve. In fact, this is almost impossible for legacy code unless the entire code is completely re-written by hand from scratch. Furthermore, a smart pointer is basically an extended data type consisting of a raw pointer and the inline metadata, i.e., a reference counter. Unlike other work using extended data types with disjoint metadata [23], a UAF defense solution based on smart pointers cannot maintain the data structure layout compatibility with the existing legacy code.

Taint Tracking. Undangle [6] utilizes taint tracking [26] to detect dangling pointers. It assigns labels to the heap pointers created from memory allocation routines and keeps track of how the pointer is copied through the registers and memory by taint tracking. Later, at memory deallocation time, it checks the labels for the pointers in the program and determine whether the pointer is unsafe based on how much time has passed since the pointer is created. Since it is based on dynamic taint tracking, it can be more precise in determining pointer locations, compared to CRCount, which relies on static type information. However, dynamic taint tracking causes significant performance overhead. It also determines unsafe dangling pointers based on the ad-hoc definition of a lifetime, which can result in an undetected UAF vulnerability.

Hardware-based Approaches. There have been several attempts to extend hardware architectures to handle UAFs efficiently. Watchdog [21] keeps disjoint metadata associated with every pointer, propagates them through the pointer operations, and checks the validity of the pointer upon every access. WatchdogLite [22] provides a fixed set of additional instructions coupled with compiler support to catch UAFs without significant hardware modifications. CHERI architecture [19] models pointers as capabilities that include information such as base and bound of the accessible memory region and distinguishes them at the hardware level so that there is no need to separately track pointers in memory as CRCount does by the means of pointer footprinting. CHERI itself does not have native support for preventing UAFs, but it does provide a foundation for accurate garbage collection.

III. Threat Model

We assume that the target C/C++ programs have UAF errors. The attacker can trigger a UAF exploit by letting a dangling pointer read/write a value from/to an object that is allocated into the same region that the previous object referred to by the dangling pointer was once allocated to. We do not consider other types of memory errors such as buffer overflow and type confusion. We assume that the integrity of the data structure and algorithm of CRCount is enforced through the security techniques that are orthogonal to CRCount [16], [15]. This assumption is consistent with previous UAF defenses relying on additional metadata [36], [17], [40], [24].

IV. Implicit Pointer Invalidation

As stated in §I, the implicit pointer invalidation scheme enables a safe, efficient defense against UAF errors, but the complications involved in reference counting hinder a wide adoption of this scheme in legacy C/C++. In this section, first, we will provide an overview of how the implicit scheme works with reference counting and then present the challenging problems to be addressed for a successful application of the scheme to real C/C++ code.

A. Invalidation with Reference Counting

In Listing 1, we present an example code to explain how UAF errors are tackled by the implicit pointer invalidation scheme coupled with the reference counters. Here, RC_{obj} denotes the reference count of a memory object *obj*. In lines 4 and 5, two heap objects, `objA` and `objB`, are created

```
1  struct node { struct node *next; int data; };
2  struct node *ptrA, *ptrB;
3
4  ptrA = malloc(sizeof(struct node));  // objA
5  ptrB = malloc(sizeof(struct node));  // objB
6
7  ptrB->next = ptrA;
8
9  /* code execution */
10
11 free(ptrA);
12
13 /* code execution */
14
15 ptrA = malloc(sizeof(struct node));
16 free(ptrB);
```

Listing 1: Code example showing the defense against UAF errors via reference counting

and pointed to by two pointer variables, `ptrA` and `ptrB`, respectively. At this moment, the reference count of each heap object is set to one. Next, `ptrA` is assigned to `ptrB->next`, and $\mathrm{RC}_{\texttt{objA}}$ is increased from one to two. Then, in line 11, the `free` function is invoked to deallocate `objA`. Now, note that $\mathrm{RC}_{\texttt{objA}} > 0$ as it is still referred to by `ptrA` and `ptrB->next`. In the explicit invalidation scheme [17], [40], [36], both the pointers are delinked with `objA` by explicitly invalidating them right after the object is deleted. However, in the implicit invalidation scheme, the further actions inside the `free` function are interrupted to leave `objA` undeleted, and the pointers remain intact, linking with the object. In line 15, `ptrA` is reassigned to point to a newly allocated object. In this case, without any explicit effort, `ptrA` is in effect invalidated with respect to `objA` because of the delinking of their reference relationship. To reflect this change, $\mathrm{RC}_{\texttt{objA}}$ is decreased from two to one. Finally, in line 16 where `objB` is freed, `ptrB->next` can also be considered to be implicitly invalidated because it is no longer legitimately accessible,[1] thus being effectively delinked with `objA`. Now, $\mathrm{RC}_{\texttt{objA}} = 0$, and thus, the object is released and can be reused safely by the memory allocator.

B. Reference Counting in C/C++

In the above example, we demonstrated how the implicit invalidation scheme with reference counting can preemptively prevent UAF errors by delaying object deletions until the reference counts are decreased to zero. Clearly, the prerequisite for this scheme is flawless reference counting, for which we developed a special mechanism to keep an accurate track of the reference relationships between the pointers and the objects along the execution flow. The reference relationship relevant to an object is expressed by the object's reference count which is dynamically increased or decreased as a pointer is linked or delinked with the object, respectively. Therefore, to accurately monitor such incessant changes in the reference count of an object, we need to pinpoint the moments at runtime when the object is linked or delinked with the pointers. We say that the referring pointers are *generated* or *killed* if the pointers are linked or delinked with their referred objects, respectively. In

[1]The pointers enclosed inside a freed object can still be accessed through a UAF vulnerability. For full security protection, these pointers must be nullified upon freeing their enclosing object.

the code, a referring pointer is generated when its value is stored in the memory, and the pointer is killed when another value overwrites the pointer (see line 15 of Listing 1) or the pointer goes out of scope (see line 16 of Listing 1). In reality, however, perfect reference counting in C/C++ is quite problematic mainly because these languages have weak typing that places no restrictions on the type conversion of objects. For instance, with weak typing, a subfield of an object can be interpreted as either a pointer or a non-pointer alternatively at the time of execution, which makes it extremely challenging to accurately capture all the generations and kills of the pointers, and accordingly update the reference counter of every corresponding referred object.

```
1  struct node { struct node *next; int data; };
2  union unode { struct node *next; int data; };
3
4  char *chunk = malloc(CHUNK_SIZE);
5  struct node *ptrA=malloc(sizeof(struct node)); //objA
6  struct node *ptrB=
7   (struct node *)&chunk[n*sizeof(struct node)]; //objB
8  union unode *ptrC=malloc(sizeof(union unode)); //objC
9
10 ptrB->next = ptrC->next = ptrA;
11
12 /* code execution */
13
14 free(ptrA);
15 ptrA = NULL;
16
17 /* code execution */
18
19 free(chunk);
20 ptrC->data = 1;
```

Listing 2: Code example showing the challenges in reference counting in a legacy C/C++ program

Listing 2 shows the practical hurdles in precise reference counting. For simplicity, we only consider $RC_{\texttt{objA}}$ in the code. There are several heap objects created in the code: `objA` and `objC` are newly allocated by `malloc` while `objB` is created by a type conversion of a subregion in the existing array `chunk`. In line 5, by linking the pointer `ptrA` with `objA`, $RC_{\texttt{objA}}$ is set to one. The pointers `ptrB` and `ptrC` are initialized to refer to `objB` and `objC`, respectively. In line 10, `ptrA` is assigned to `ptrB->next` and `ptrC->next`, which results in $RC_{\texttt{objA}} = 3$. In line 14, the programmer wants to delete `objA` when $RC_{\texttt{objA}} > 0$, but as mentioned earlier, this deletion will be denied. In the next line, where `ptrA` is assigned NULL, $RC_{\texttt{objA}}$ is decremented by one. The last two lines of the code exhibit two challenging problems pertaining to reference counting. Firstly, when the array `chunk` is deleted, $RC_{\texttt{objA}}$ has to be decreased as `objA` is referred to by a pointer, `ptrB->next`, which is inside the deleted object. Unfortunately, as `chunk` is declared as an ordinary array, the compiler cannot provide any information with regard to the existence of a pointer inside at runtime. Therefore, for correct reference counting, we need some mechanism to separately track the location of the pointers inside `chunk`. The code in line 20 presents another practical problem. Here, when `ptrC->data` is set to 1, the previously stored pointer, `ptrC->next`, is simultaneously overwritten by the same value. Therefore, according to the implicit invalidation scheme, $RC_{\texttt{objA}}$ should be reduced as the pointer referring to the object has been technically killed. Here, for correct reference counting, we need to analyze the code and mark the point so that we can decrease the reference count at runtime, and we also need to track whether the pointer is currently stored at the location of `ptrC->data` at that moment.

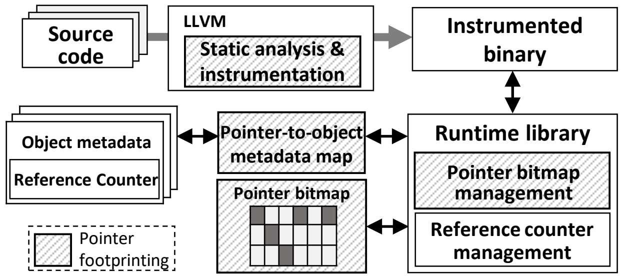
Source code
LLVM
Static analysis & instrumentation
Instrumented binary
Object metadata
Reference Counter
Pointer-to-object metadata map
Runtime library
Pointer bitmap management
Pointer bitmap
Reference counter management
Pointer footprinting

Fig. 1: Overview of CRCount

From all these examples, we can conclude that without a detailed tracking down of all the operations affecting the generations and the kills of the referring pointers, the accuracy of reference counting would be severely limited. This would in turn damage the overall feasibility of the implicit invalidation scheme for mitigation against UAF errors. To summarize, as hinted by the examples, the identification of all the pointer generations and kills in the legacy code is prohibitively complex. The failure to find some pointer generations will result in underestimated reference counts, inducing loopholes in the mitigation of UAF errors. The opposite (i.e., failure to find the kills) will lead to overestimated counts, which, in turn, will result in memory leaks. In the subsequent sections, we will show how CRCount addresses this challenge.

V. Design

In this section, we elaborate on the design of CRCount, our UAF error prevention system based on implicit pointer invalidation. First, we present a brief overview of the entire system, and then we provide a more detailed explanation of each component.

A. Overview

Figure 1 shows an overview of CRCount. At the core, CRCount uses a technique called *pointer footprinting* (§V-B) to overcome the challenge described in §IV. The pointer footprinting technique tracks exactly when and where in memory the pointers to heap objects are generated and killed. This technique is centered around a special data structure, the *pointer bitmap*, that represents the exact locations of the heap pointers scattered throughout the program memory. The bitmap is managed by the *runtime library*, which keeps track of the generations and the kills of the pointers at runtime, and reflects the changes into the bitmap by setting or clearing the corresponding bits, respectively. Invocations to the runtime library are instrumented into the target program by CRCount's *LLVM plugin*, which utilizes a static analysis to minimize the number of instrumentation points while preserving the precision in tracking pointers. The idea of using the bitmap to indicate pointer locations has been proposed in previous work on garbage collection [27], [34]. However, unlike previous work, with the help of the compiler instrumentation and the runtime library, we automatically and accurately track down the heap pointers in the entire memory space to enable successful mitigation of UAF errors.

CRCount depends heavily on the pointer footprinting technique for precise reference counting. It associates each heap object with per-object metadata (§V-C1) that include the reference counter for the object. Every generation or kill of a pointer is detected and handled by the runtime library. CRCount takes the stored/killed pointer value and consults with the pointer-to-object metadata map to find and increase/decrease the reference count of the object referred to by the pointers (§V-C2). When the `free` function is called to deallocate an object, CRCount first checks the object's reference counter. If the count is zero, then CRCount lets the function deallocate the object. Otherwise, it halts the function and leaves the object intact. Later, when there is a change (either increment or decrement) in the reference count, CRCount kicks in and checks whether the count is zero. Finally, when the count decreases to zero, implying that the pointers having referred to the object are all implicitly invalidated, CRCount hands the object over to the memory allocator, which will free and reuse the object.

B. Pointer Footprinting

To enable the precise tracking of heap pointers, CRCount uses the pointer footprinting technique, which is centered around the pointer bitmap data structure that enables us to efficiently locate all the pointers in the memory that refers to the heap objects. The pointer bitmap is basically a shadow memory for the entire virtual memory space, which marks the locations where the heap pointers are stored. We assume that pointers are aligned to an 8-byte boundary, which would be true in most cases as pointer-type variables are typically arranged in an 8-byte alignment by the compiler in a 64-bit system.[2] Note here that the current prototype of CRCount only supports a 64-bit system. Based on this assumption, each bit of the pointer bitmap corresponds one-to-one to all the 8-byte-aligned addresses in the virtual memory space; thus, we can identify the exact pointer locations through the pointer bitmap. Owing to the structural simplicity and compactness of our bitmap, the runtime library can efficiently manipulate it with a combination of simple bit operations such as shifting and masking. The bitmap occupies 1/64-th of the virtual memory space and is reserved at the start of the process through the `mmap` system call. This might seem like a large amount of memory, but fortunately, because of the demand paging mechanism of OSs that delays the allocation of a physical memory block (i.e., frame) until there is an actual access, the bitmap does not occupy much physical memory at runtime. Furthermore, as the access to the pointer bitmap follows the original locality of the memory accesses, in practice, the physical memory overhead for the bitmap is negligible.

The pointer bitmap is managed by the runtime library. Table I shows a list of the runtime library functions, along with the program points where they are invoked and their tasks at these points. The function `crc_alloc` does not update the pointer bitmap, but when a new heap object is allocated, it

[2]We have encountered a few cases where this assumption does not hold true. We will explain these cases in §IX.

Runtime library function	Invoked at	Description
`crc_alloc`	Heap allocation	Add a mapping for the new heap object to the pointer-to-object metadata map
`crc_store`	Candidate store Instruction	Handle a pointer generation and/or kill due to memory store
`crc_memset`	Memset	Handle pointer kills due to memset'ing a region with identical bytes
`crc_memcpy`	Memcpy	Handle pointer generations and/or kills due to copying of a memory region
`crc_free`	Heap deallocation	Handle pointer kills by heap object deallocation
`crc_return`	Function return	Handle pointer kills by stack frame deallocation

TABLE I: The list of runtime library functions of CRCount

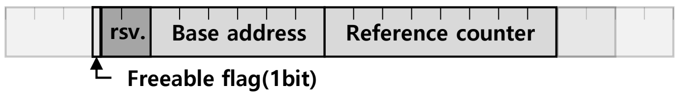

Fig. 2: Layout of per-object metadata. `rsv.` field is reserved for C++ support (§VI) and garbage collection (§VII).

adds a new mapping for the object to the pointer-to-object metadata map to be used in the reference count management (refer to §V-C1 for details). Moreover, as we are only interested in the pointers to the heap objects, the runtime library functions look up the pointer-to-object metadata map before setting the bits in the pointer bitmap. The function `crc_store` sets or clears, respectively, the corresponding bit in the bitmap when a new heap pointer is stored (generated) or the previously stored pointer is overwritten (killed) by a store instruction. The functions `crc_memset` and `crc_memcpy` set and clear the bits in the pointer bitmap corresponding to the pointers that are killed and/or duplicated by `memset` or `memcpy`. The functions `crc_free` and `crc_return` clear the bits in the pointer bitmap corresponding to the pointers invalidated by the heap object deallocation and the stack frame deallocation, respectively.

At the time of compilation, the calls that invoke the runtime library functions are instrumented into the program so that the runtime library can reflect the generations and the kills of the pointers into the pointer bitmap. The instrumentation is done by the CRCount's LLVM plugin that provides an additional pass over the intermediate representation (IR) during the compilation phase. All the runtime library calls except `crc_store` are instrumented in a straightforward manner at every corresponding program point. In the case of `crc_store`, instrumenting all the store instructions will cause the excessive performance overhead. It will be overkill if we consider that only a part of these instructions are actually related to pointer generations and kills. However, as the store of a non-pointer-type value can kill a pointer, as discussed in §IV-B, a simple examination of the type of stored value in LLVM IR is not sufficient to identify all the instructions that need to be instrumented. To solve this, our LLVM plugin performs a static analysis of the program code to identify the minimum set of instrumentation points required to enable an efficient yet precise tracking of pointers.

Listing 3 shows the pseudo-code of CRCount's LLVM plugin for instrumenting memory store instructions. In the LLVM IR, store instructions assign a source value `val` to a destination

```
for storeInst in program:
  dest = storeInst.dest
  val = storeInst.val

  if !isPointerType(val) && !isCastFromPtr(val):
    if !shouldInstrument(storeInst.dest):
      continue

  if isLoadStoreSame(dest, val):
    continue

  callInst = createCallInst(crc_store, dest, val)
  storeInst.insertBefore(callInst)
```

Listing 3: Pseudo code for instrumenting the store instructions.

address `dest`. We should definitely insert a `crc_store` call when a *pointer* value is written; therefore, the plugin first examines `val` to check whether it is a pointer. It is obviously a pointer if it has a pointer type (`isPointerType`), but sometimes, it can be a pointer even if it does not have a pointer type. For example, the programmer could have cast a pointer into an integer type. In this case, in the IR code, there will be a `bitcast` instruction that casts the type of `val` somewhere before the store instruction. In this context, our LLVM plugin conducts a backward data flow analysis to check whether `val` has been cast from a pointer type prior to the store instruction (`isCastFromPtr`). If it has, then the store instruction is instrumented.

Even if `val` is not a pointer type value, store instructions might implicitly invalidate an existing pointer by overwriting it with a non-pointer value. Thus, the LLVM plugin performs a backward data flow analysis on `dest` to check whether the store instruction can potentially kill a pointer and thus should be instrumented with a call to `crc_store` (`shouldInstrument`). There are two main cases where the instrumentation is necessary. First, the plugin instruments the store instruction if `dest` has been cast from a double pointer type, because in this case, the memory pointed to by `dest` can hold a pointer value. Another case that the plugin mainly looks for in the data flow analysis is a case wherein `dest` is a field of the union type that can hold both a pointer value and a non-pointer value (as shown in Listing 2). However, the determination of whether or not a specific field of the union can be a pointer type in LLVM IR is non-trivial because the IR code generation phase collapses the type information for the union type, and thus, union types in LLVM only has the type information for a single member field whose in-memory representation is the largest in size among all the fields in the union. For example, if a union type has a pointer type member and a struct type member with the size bigger than that of a pointer, only the struct type is shown as the member of the union type in IR. Nevertheless, with the backward data flow analysis, we can at least determine whether the field pointed to by `dest` is a part of a union type. Consequently, we conservatively instrument the store instruction if the underlying type of the memory object is a union type even if it does not have a pointer type member field at the specific offset.

The LLVM plugin also performs a similar optimization done in DangSan that skips the instrumentation if it can be statically determined that `val` points to the same object that the pointer stored in `dest` points to (`isLoadStoreSame`). In this case, `crc_store` will increment and decrement the same reference counter, so there is no need for the runtime library call to be instrumented. This mainly deals with the case where a pointer is simply incremented or decremented and thus the reference counter of the target object does not change.

C. Delayed Object Free

To achieve its objective, CRCount enforces the *delayed object free* policy that delays the freeing action as briefed in §V-A. CRCount manages the reference counters of objects by using the pointer footprinting technique. When a programmer invokes the function `free`/`delete` to free an object, CRCount checks the object's reference count and stops the function from freeing the object if this count is non-zero. To implement this, we modified the `free` function so that the function cannot automatically free objects. In CRCount, the decision on when to free an object is exclusively made by our runtime library. Therefore, any manual attempt of a programmer to delete an object is intercepted by the library which will eventually permit the memory allocator to free the object for reuse when the object's count becomes zero.

1) Per-object Metadata: To realize the delayed object free policy, we must maintain a reference counter for each heap object. To do this, CRCount uses METAlloc [11] to augment the heap objects with the per-object metadata. METAlloc internally maintains a trie-based pointer-to-object metadata map [25]. Given a pointer value, METAlloc retrieves the map and returns a pointer to the object metadata allocated separately when the heap object is allocated. The per-object metadata (Figure 2) include not only the reference counter but also two additional pieces of information: the *base address* and a 1-bit *freeable flag*. The base address is required for the memory allocator to free the object when the reference count becomes zero. Note that the `free` function needs the base address of the target object as its unique argument. However, when the last pointer that points to the object is killed, and the reference count is set to zero, there is no guarantee that this pointer will hold the base address of the object. Therefore, CRCount separately keeps the base address of each object to invoke the `free` function correctly. The freeable flag is required for CRCount to mark some objects as *freeable*. When the `free` function is called for an object and its reference count is non-zero, CRCount just halts the function and sets the freeable flag of the object. Thereafter, when the reference counts of objects become zero, CRCount allows only the freeable objects for which the `free` function has been called, to be actually freed by the memory allocator. This is important for CRCount because there are some exceptional cases (discussed in detail in §IX) that may hinder the correct maintenance of the reference counter. These exceptional cases would decrease even the reference counters of non-freeable objects to zero, and if CRCount mistakenly decides to free these non-freeable objects, the program may crash. Even though such cases are known to be unusual in the normal programming practices of C/C++ [30], we adopt this freeable flag-based approach for maximum compatibility with the legacy C/C++ applications.

2) Reference Counter Management: The runtime library includes the code for reference counter management that can update the reference counter according to the pointer generations and kills. When a heap object is allocated, the associated per-object metadata are also allocated. Here, the reference count is initialized to zero, and the base pointer is set to the base address of the allocated memory region. Every time a pointer is stored by a store instruction, CRCount reads the corresponding per-object metadata by using the pointer-to-object metadata map and increases the reference count. For `memcpy`, CRCount first examines the pointer bitmap mapped to the source memory region to find the pointers that are to be duplicated and increases the reference counts corresponding to the objects referred to by these pointers. Every time a pointer is invalidated, either by a store instruction or by any of the `memset`/`memcpy`/`free`/`return` function/instruction, CRCount checks the pointer bitmap to identify the pointers from the destination memory region and decreases the reference counts of the objects referred to by these pointers. For `free` and `return`, CRCount also nullifies all the pointers inside an object or a stack frame to completely block wrongful uses of them. Finally, when CRCount finds that the reference count for an object has become zero and the object's freeable flag is set, it gives the object to the memory allocator that will free the object.

It is noteworthy that CRCount handles `memset`/`memcpy` as well. Not only are they very commonly used in C/C++ programs, but they are also often introduced by compiler optimizations when a contiguous range of memory is set or copied. The previous work on pointer invalidation, such as FreeSentry or DangSan, does not handle these functions for performance reasons, leaving the system exposed to UAF errors. Note that CRCount is immune to the so-called *reference cycle* problem [37]. Automatic memory management systems (i.e., garbage collector) relying on reference counting suffer from the problem wherein the reference counters of a group of objects are never decreased to zero when the objects are cross-referenced. Since the purpose of automatic memory management systems is to deallocate memory objects automatically without relying on explicit free requests, the reference counts of objects pointing each other will never decrease to zero. To avoid this problem, many systems introduce the notion of *weak* references, which the programmers must wisely use to prevent reference cycles [18], [29]. CRCount does not suffer from reference cycles as it operates based on the `free` functions that already exist in the legacy code. When the `free` function is called for one of the objects involved in the reference cycle, CRCount forcibly kills the pointers enclosed in the freed object and decrements the reference counter of the other object, thereby breaking any reference cycles.

VI. Implementation

We have implemented the CRCount LLVM plugin as an LTO (Link Time Optimization) module based on LLVM 3.8. The runtime library is written in C and is statically linked into the program binary. The LLVM plugin and the runtime library each consists of approximately 1k lines of code.

Allocation of the per-object metadata. METAlloc provides an efficient mapping between a given pointer and the associated per-object metadata, but it does not provide any way to allocate the metadata itself. We sought for a way to avoid the additional overhead that comes from metadata allocations, since whenever heap object is allocated, the corresponding per-object metadata also needs to be allocated. If we use `malloc` for this purpose, an overhead incurred by `malloc` would be doubled, which could be non-negligible as more memory objects are allocated [10]. Fortunately, each of our per-object metadata mapped to the objects has a fixed size. Thus we can mitigate the metadata allocation overhead by using the concept of an object pool. We first reserve an object pool using `mmap` and provided a custom allocator for the per-object metadata, eliminating the costs involved with `malloc`. The current implementation of CRCount performs a linear search over this memory pool to find an empty slot for the allocation of the metadata.

Handling realloc. `realloc` can migrate an object from its original memory region to another memory region. Such behavior of `realloc` necessitates an exceptional handling by CRCount. First, when the contents of the target object are copied to another region, the pointers belonging to the object are copied as well. Therefore, to keep track of the copied pointers correctly, the corresponding bits of the pointer bitmap also have to be copied. Next, after the migration, `realloc` frees the original memory region. In CRCount, however, the free action only has to be allowed when the reference count becomes zero. To enforce this rule, we modified `realloc` to let the runtime library decide when to free the original region, as was done in the `free` function. The runtime library (1) allows the memory allocator to perform the free action if the reference counter is zero or (2) just sets the freeable flag otherwise.

Multithreading support. Multithreading support can be enabled in CRCount by defining `ENABLE_MULTITHREAD` macro variable when building the runtime library. Two major data structures—the reference counters and the pointer bitmap—have to be updated atomically to support multithreading. The reference counters need atomic operations because multiple threads can store or kill the pointers to the same heap object at the same time. As a reference counter is just a single word, we simply used the atomic operations defined in the c11 standard library. We assume that the threads in the target program do not write to the same pointer concurrently without a proper synchronization. We believe that this is a reasonable assumption as it indicates a race condition in the original program. The pointer bitmap also must be maintained in an atomic fashion. Even if we have the above assumption, multiple threads could write pointers to the nearby memory locations which could end up in the same word in the pointer bitmap. Thus, we also use the atomic operations whenever the bitmap is updated. Besides the reference counters and the pointer bitmap, we made a small change in the per-object metadata allocation/deallocation routine to ensure thread safety.

Note that all of the data structure updates in CRCount only require touching just one word which makes multithreading support very simple and also very efficient in most cases. However, we have encountered a worst case in one of the benchmarks that we tested, where only a small number of objects are allocated and their reference counters are frequently updated by multiple threads. In this case, there will be many lock contentions for the reference counters, which results in a considerable performance overhead. We will give more detail in §VII.

Double free and invalid free. We can simply implement the prevention capability for double frees on CRCount. As a freeable flag of per-object metadata indicates that `free` has been called for an object, we can easily detect if `free` is called multiple times for the object. CRCount can also be extended to prevent invalid frees. If `free` is called for an invalid pointer, CRCount can easily detect it because there either will be no valid mapping for the pointer in the pointer-to-object map, or the base address of the object metadata will not match the pointer value.

C++ support. For the most part, CRCount can naturally support C++ because CRCount instrumentation operates on LLVM IR, which is language independent and thus does not distinguish between C and C++. C++ concepts like templates, dynamic binding, etc. are lowered to basic functions and LLVM instructions and do not require separate handling by our LLVM plugin. However, C++ `new` and `delete` require some special care. Recall that CRCount delays freeing of the object until its reference count becomes zero. For C++, CRCount must invoke the correct deallocation function according to the function that was used to allocate the object. `malloc`, `new`, and `new []` are three possible choices for the allocation of the object, and the corresponding deallocation function must be used to deallocate the object. To achieve this, CRCount uses the additional bits next to the freeable flag in the per-object metadata to record and call the right function for the deallocation of the object.

VII. Evaluation

In this section, we evaluate CRCount by measuring the performance overhead and the memory overhead imposed by CRCount in well-known benchmarks and web server applications. All the experiments have been conducted on Intel Xeon(R) CPU E5-2630 v4 platform with 10 cores at 2.20 GHz and 64 GB of memory, running Ubuntu 64-bit 16.04 version. We applied minor patches to a few of the benchmarks to assist our reference counter management. In §IX, we will explain these cases in detail.

A. Statistics

The performance and memory overhead of CRCount can vary depending on the characteristics of the target program. In particular, the number of pointer store operations and the memory usage of CRCount can be a crucial indicator for analyzing the experimental results. Thus, we gathered some of the statistics for the SPEC benchmarks [12] which we will refer to when analyzing the performance and memory overheads in this section. Table II shows the results for the SPEC CPU2006 benchmarks.

Here, we first compare the number of pointer stores tracked down by CRCount with that by DangSan. We will explain other metrics later in this section. As shown in the `# ptr stores by inst.` column and the `# ptrs` column, in many benchmarks, we can see that the number of pointer stores by the store instruction measured in CRCount is larger than the one in DangSan. The differences are mainly due to a small patch we applied to LLVM in order to ease our static analysis. Specifically, we disabled a part of the bitcast folding optimization, which complicates our backward data flow analysis in tracing the casting operations. We expect the numbers to be decreased if we elaborate on our static analysis to support the optimization, which would also give a small performance improvement for CRCount.

In the case of `dealII` and `xalancbmk`, CRCount kept track of a fewer number of pointer stores than that of DangSan. This is due to a minor hack in our LLVM plugin that is applied to avoid the problem of incorrect reference counter management in the programs that use the C++ templates from the C++ standard library. Specifically, the problem occurred because only the part of the library code for the template functions defined in the header file was instrumented by our plugin while the rest was not instrumented. In order to solve this problem, we compiled the program with the `-g` option to include the debug symbols and excluded the instructions originating from the library during the instrumentation. Another way to solve this problem would be to compile and instrument the entire standard library with CRCount.

B. Performance Overhead

To measure the performance overhead of CRCount, we ran and recorded the execution times for several benchmarks and server programs. We compare the performance overhead of CRCount with DangSan and Oscar, which are the latest work in this field. We used the open-sourced version of DangSan for our evaluation while using the numbers reported in the paper for Oscar. We also report the performance overheads for BDW GC. To use BDW GC, programs must use special APIs for memory allocation routines (e.g., GC_malloc instead of malloc) to let the GC track and automatically release the object when there are no references to it. BDW GC provides an option to automatically redirect all of the C memory management functions to use the APIs. We used this option and another option that makes GC to ignore the *free* function. As we later specify, we were not able to compile or correctly run some C application, which indicates that some porting efforts are required to use BDW GC for UAF mitigation. Also, simple API redirection does not work for C++ applications. Instead, all the class needs to inherit from special `gc` class which provides a new definition of `operator new`. Classes that already have a custom `operator new` function will have to be changed. Because of these reasons, we only show the results for the subset of the C benchmarks which we were able to compile and run correctly.

First, to measure the performance impact on the single-threaded applications, we ran the SPEC CPU2006 benchmark suite [3]. Figure 3 shows the results. For CRCount, the geometric mean of all benchmarks is 22.0%, which is approximately the half that for DangSan and Oscar, which respectively are 44.4% and 41%. The performance efficiency of CRCount is even more evident in the pointer intensive benchmarks (`omnetpp`, `perlbench`, and `xalancbmk`). For these benchmarks, CRCount only incurs an average overhead of 92.0%, while both DangSan and Oscar show over 300%. For `povray`, CRCount incurs a higher performance overhead than Oscar and DangSan. For the case of Oscar, note that Oscar does not instrument any memory access. This gives Oscar performance advantages for some benchmarks like

[3]linked with the single-threaded version of CRCount runtime library

benchmark	CRCount							DangSan
	# tot alloc.	# ptr stores by inst.	# ptr stores by `memcpy`	max mem.	max undeleted	max undel. / max mem.	leaks	# ptrs
400.perlbench	350m	44507m	242m	1103 MB	5838 KB	0.005	1680 B	40490m
401.bzip2	264	2200k	0	3362 MB	0	0	0	2200k
403.gcc	28m	9328m	13m	4075 MB	7491 MB	1.838	288 KB	7170m
429.mcf	21	10086m	574k	1676 MB	0	0	0	7658m
433.milc	6531	2663m	0	679 MB	21 MB	0.032	0	2585m
444.namd	1340	2998k	1198	46 MB	0	0	0	2970k
445.gobmk	622k	609m	10	117 MB	34 KB	0	0	607m
447.dealII	151m	30m	13m	791 MB	2048 KB	0.003	0	117m
450.soplex	236k	876m	1731k	877 MB	27 MB	0.032	0	836m
453.povray	2427k	5784m	2409m	2 MB	18 KB	0.007	0	4679m
456.hmmer	2394k	4458k	0	41 MB	12 MB	0.291	0	3829k
458.sjeng	21	4	0	172 MB	0	0	0	4
462.libquantum	165	130	72	96 MB	32 B	0	0	130
464.h264ref	178k	11m	845m	111 MB	1609 KB	0.014	0	11m
470.lbm	20	6004	0	409 MB	0	0	0	6004
471.omnetpp	267m	13099m	14m	154 MB	1301 KB	0.008	481 KB	13099m
473.astar	4800k	1235m	7667k	471 MB	91 MB	0.195	0	1235m
482.sphinx3	14m	326m	0	44 MB	11 KB	0	0	302m
483.xalancbmk	135m	1711m	1633	385 MB	1018 KB	0.003	576 B	2387m

TABLE II: Statistics for the SPEC CPU2006 benchmarks. `# tot alloc.` denotes the total number of object allocations. `# ptr stores by inst.` denotes the number of tracked pointer stores by the store instructions, while `# ptr stores by memcpy` denotes the number of pointer stores by `memcpy`. `max mem.` shows the maximum amount of memory occupied by the objects that are allocated but not freed. `max undeleted` shows the maximum amount of memory occupied by the undeleted objects. `max undel. / max mem.` shows the ratio between `max mem` and `max undeleted`. `leaks` shows the memory leak caused by an error in the pointer footprinting. The last column shows the number of pointers tracked down by DangSan.

`povray` which have relatively a large number of pointer stores (see Table II). For the case of DangSan, let us note that DangSan does not track down any pointers copied through `memcpy`. In contrast, CRCount does track down such pointers for higher accuracy (thus also security), which explains the larger performance overhead of CRCount. For `dealII` and `xalancbmk`, we should consider the advantage that CRCount might obtain by not instrumenting the template-based standard library functions. However, considering the difference between the number of tracked pointers described in Table II, we still expect that the performance overhead of CRCount would be lower than those of DangSan and Oscar. For BDW GC, we could not run `gcc` benchmark. The geometric mean of the performance overhead for the rest of the C benchmark is 0.7% for BDW GC and 13.9% for CRCount, which shows that the current highly optimized and multi-threaded BDW GC can be very efficient for single-threaded workloads compared to CRCount which suffers from instrumentation overheads.

We also conducted a set of experiments with the PARSEC [4] benchmarks to evaluate the scalability of CRCount in multithreaded programs. Figure 5 shows the results in comparison to the baseline and DangSan. The geometric mean of the overheads (excluding `freqmine`) ranges from 6.1% to 22.4% in CRCount and from 6.3% to 17.0% in DangSan, as more threads run concurrently. Overall, CRCount and DangSan show comparable performance overhead in most of the benchmarks. Even though CRCount uses atomic operations to maintain its data structures, it does not introduce critical sections because only a single word needs to be updated at a time. Also, simultaneous accesses to the same reference counter or the same word in the pointer bitmap are rare. Thus, CRCount can be scaled to multiple threads in most cases. `barnes` shows an interesting behavior as it is run with more threads. In `barnes`, only a few large objects are allocated with around 6 billion pointer stores. As the total number of objects is so small, we expect that frequent lock contentions occur when updating the reference counts, which explains such an irregular result. For the subset of the benchmarks we could test with BDW GC, the geometric mean of the overheads

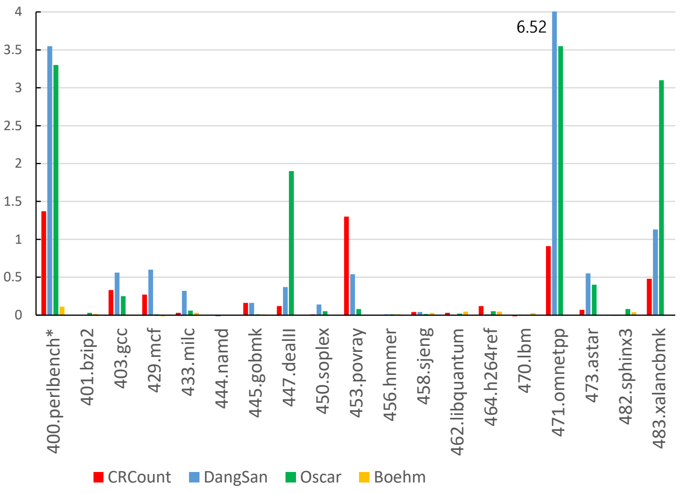

Fig. 3: Performance overhead on SPEC CPU2006. We use the reported numbers in the original papers for `perlbench` of DangSan, which fails to run, and all the benchmarks of Oscar. For Boehm GC, we were able to run only C benchmarks excluding `gcc`.

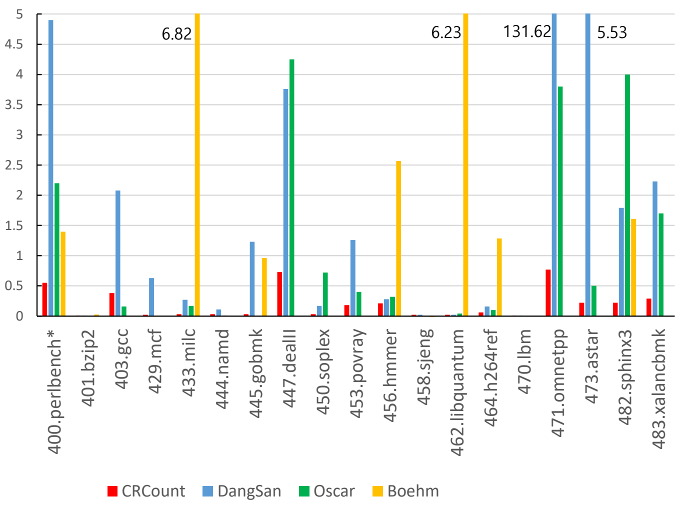

Fig. 4: Memory overhead on SPEC CPU2006. Some numbers are those that have been reported in the original paper as in Figure 3.

ranges from 5.3% to 28.9% in BDW GC and 4.9% to 28.6% in CRCount. CRCount performs comparable to BDW GC for multithreaded workloads.

We conducted additional experiments for evaluating the performance of CRCount on web server applications, including Apache 2.4.33 (with `worker MPM`), Nginx 1.14.0, and Cherokee. We tested each web server with the default configuration files through Apachebench (with 128 concurrent connections and 1,000,000 requests), and measured the throughput in terms of requests per second (RPS). For Apache, the throughput of the baseline is 24024 RPS, while it is decreased to 23051 RPS (slowdown of 4.1%) in CRCount and 22774 RPS (slowdown of 5.2%) in DangSan. The results for other web servers are similar. For Nginx, the throughput of the baseline was 29514 RPS, but it is 20553 RPS (slowdown of 30.4%) in CRCount and 20144 RPS (slowdown of 31.7%) in DangSan. Lastly, for Cherokee, the baseline throughput of 25993 RPS is decreased to 25615 RPS (slowdown of 1.5%) and 24756 RPS (slowdown of 4.8%) in CRCount and DangSan, respectively.

C. Memory Overhead

In CRCount, let alone its data structures, undeleted objects may be one major factor that potentially consumes substantial memory. To evaluate the overall memory overhead of CRCount, we have recorded the maximum resident set size (RSS) while running the same benchmarks as in §VII-B.

Figure 4 shows the memory overhead of our CRCount, DangSan, Oscar, and BDW GC for SPEC CPU 2006 benchmarks. Our geometric mean of all benchmarks is 18.0%, which is significantly lower than 126.4% of DangSan and 61.5% of Oscar. BDW GC shows a memory overhead of 125.7% for the tested benchmarks while that of CRCount is 9.7%. Figure 6 shows the maximum RSS values for PARSEC benchmarks for baseline, CRCount, DangSan, and BDW GC. The geometric mean (without `freqmine`) of the overhead is from 9.2% to 11.6% in CRCount and from 45.0% to 52.7% in DangSan as the number of threads increases from 1 to 64. The geometric mean of the memory overhead for the benchmarks tested with BDW GC ranges from 56.6% to 70.9% for BDW GC and 5.4% to 6.0% for CRCount.

Finally, we measured the memory overhead for three web server applications used in §VII-B. The maximum RSS of Apache is 7.8MB in the baseline, 9.9MB in CRCount (26% overhead), and 106.8MB in DangSan (1263% overhead). For Nginx, the maximum RSS is 6.0MB in the baseline, 6.5MB in CRCount (8.2% overhead) and 10.4MB in DangSan (73.3% overhead). For Cherokee, the recorded maximum RSS is 32.1MB, 41.2MB (28.5% overhead) and 62.9MB (95.9% overhead), in the baseline, CRCount and DangSan, respectively.

All those experimental results, we believe, consistently testify the efficiency of CRCount in terms of memory usage. Such memory efficiency of CRCount would be attributed to its compact data structures, but more importantly, to the relatively low memory usage by undeleted objects that remains persistently small in practice. To investigate the relative overhead of undeleted objects further, see Table II where the `max mem.` and `max undeleted` columns respectively show the maximum total memory for the heap-allocated objects and the undeleted objects. In the `max undel./max mem.` column, we compute the relative overhead of undeleted objects in memory, which is clearly shown to be very small for most benchmarks. On top of that, we have discovered that the majority of these undeleted objects tend to be eventually deleted and handed over by CRCount to the allocator for safe reuse during program execution. We credit such favorable outcomes mainly to the capability of CRCount that is able to correctly decrease the reference counts whenever generated pointers are killed.

There are still the cases where CRCount fails to accurately keep up the reference counts, thereby being unable to delete undeleted objects even when no more pointers refer to them (see §IX). The `leaks` column in Table II denotes the total amount of memory occupied by such undeleted objects. To calculate the numbers in the column, right after program termination, we scanned the entire pointer bitmap to decrease the reference counters corresponding to the pointers still residing in the global variables or the heap objects for which the `free` function has not been called during the execution. The existence of the undeleted objects that still have a non-zero reference count after this process signifies that some pointer kills were not tracked properly, failing to decrease the reference count of these objects. Note that once CRCount fails to track a pointer kill, it is no longer able to delete the corresponding object as the reference count of the object will never decrease to zero. Obviously, these objects are the source of the memory leak induced by CRCount. Luckily, we can see that the numbers on the `leaks` column are negligibly small (or even zero) for almost all benchmarks, indicating that CRCount in fact quite accurately perform reference counting in legacy C/C++ code.

The numbers in Table II only inform us of the maximum memory space that has once been occupied by heap and undeleted objects during program execution, but it does not give us any clue how much space has been dynamically consumed by these objects at runtime. To obtain this, we have regularly measured the changes in the amount of the memory taken up by undeleted objects and memory leaks

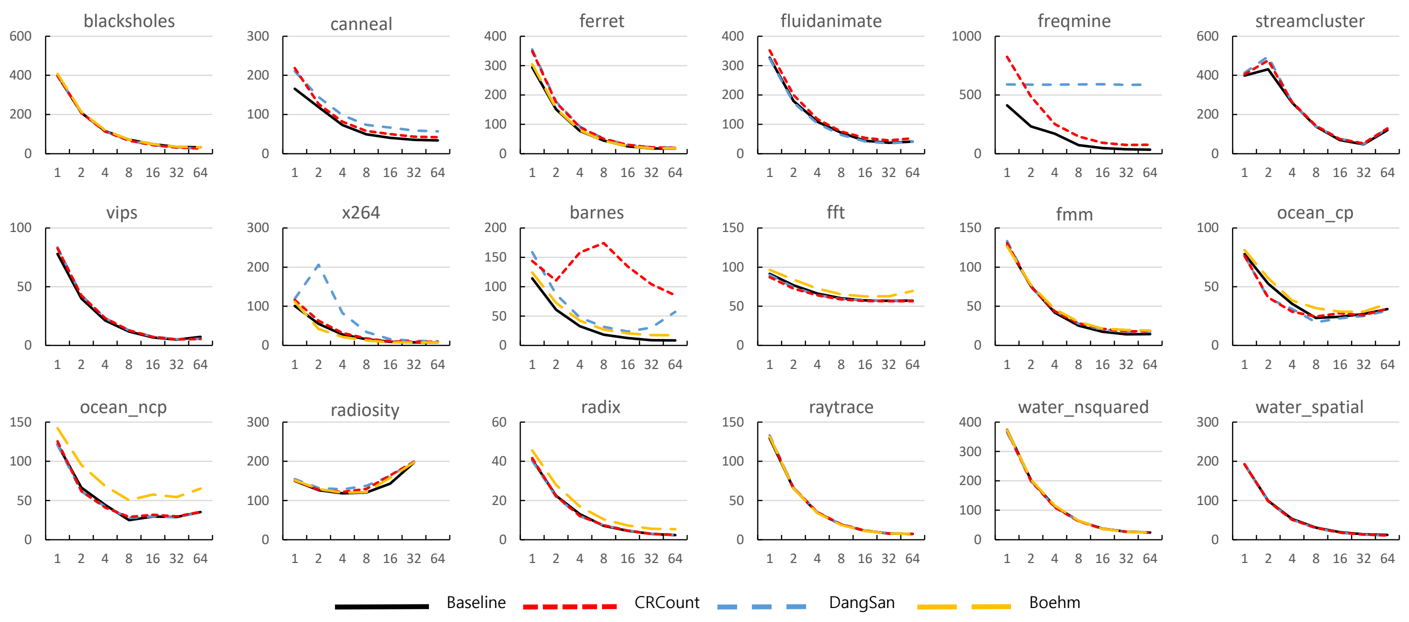

Fig. 5: Comparison of the execution time on PARSEC. We could not get the correct result for `freqmine` for DangSan because we could not enable OpenMP with DangSan, which is required to run `freqmine` in the multithreaded mode. The results for Boehm GC is only included for the subset of the C benchmarks that we could run.

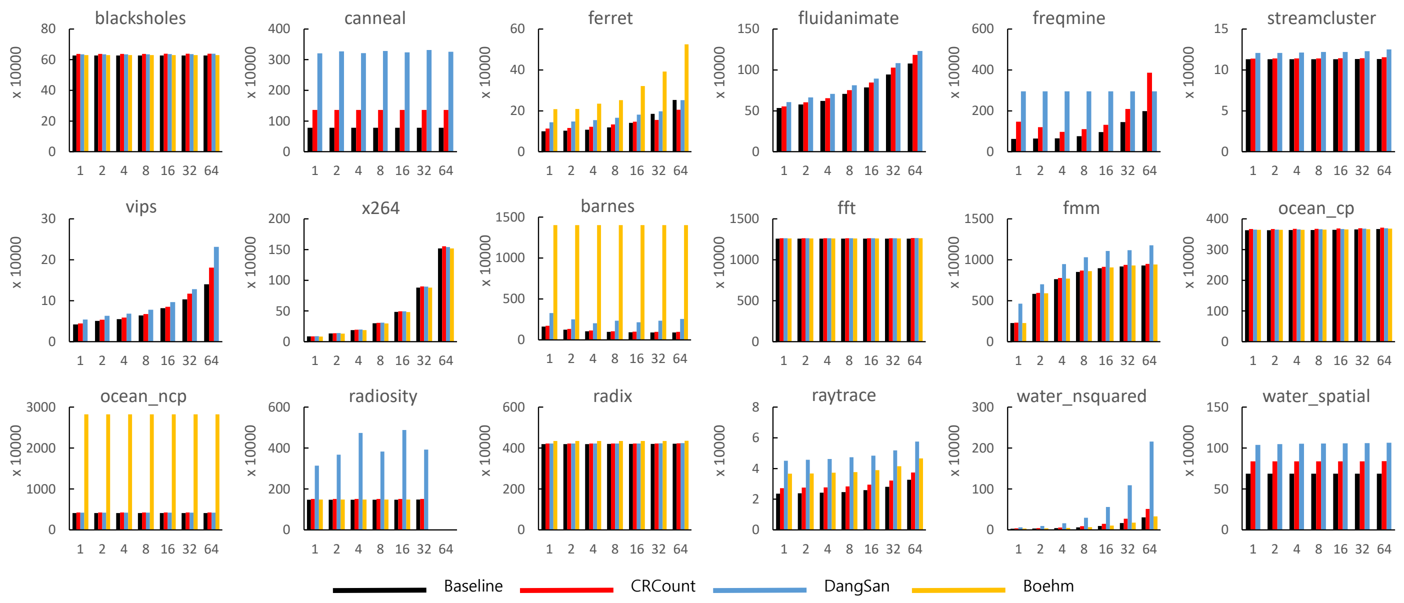

Fig. 6: Memory overhead on PARSEC.

Application	CVE	Vulnerability	Original	CRCount	CRCount-det
openlitespeed-1.3.7	2015-3890	UAF	No effect	No effect	Detected UAF
wireshark-2.0.1	2016-4077	UAF	No effect	No effect	Detected UAF
PHP-5.5.9	2016-3141	UAF	Crash (double free)	Detected double free	Detected UAF
PHP-5.5.9	2016-6290	UAF	No effect	Detected double free	Detected UAF
PHP-5.5.9	2016-5772	Double free	Crash (double free)	Detected double free	Detected double free
ed-1.14.1	2017-5357	Invalid free	Crash (invalid free)	Detected invalid free	Detected invalid free

TABLE III: Real world vulnerabilities tested with CRCount. The `Original` column shows the behavior of the original program when run with the exploit input. We disabled the `Zend` allocator in `PHP` to test the exploits.

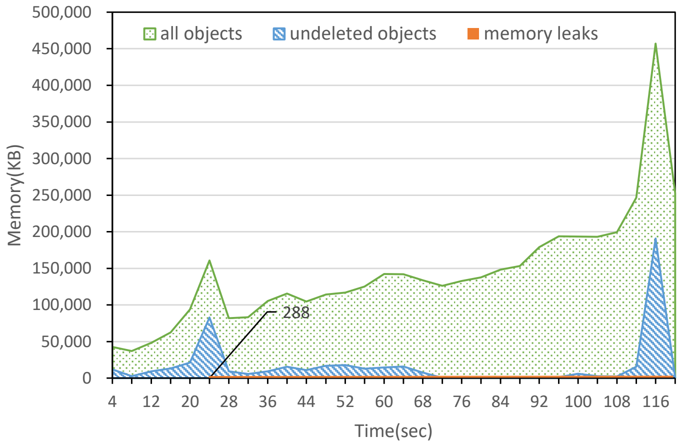

Fig. 7: Changes in memory usage during the execution of `gcc` with `200.i` input file. `all objects` denotes the total amount of memory allocated for heap-allocated objects and the undeleted objects. `undeleted objects` and `memory leaks` indicate the amount of the memory occupied by undeleted objects and memory leaks, respectively.

over the entire period of each benchmark executions. As can be expected from II, in most benchmarks, the total memory overhead due to the undeleted objects steadily remains low throughout the execution. However in some benchmarks like `gcc` with `200.i` input file (see Figure 7), the overhead can sometimes become noticeably high at some point during program execution although it remains low for most of the execution times. Figure 7 displays two peaks in the memory consumption when a large amount of memory is consumed by undeleted objects, but most of it is soon freed as the result of program's normal execution. Figure 7 also displays the amount of leaked memory. Note that once a memory leak occurs at some point in the execution, it will never disappear afterward. For instance in Figure 7, we have a memory leak of 288 KB in the middle of execution which exists until the end of execution. Fortunately, the amount of wasted memory due to memory leaks is negligible in comparison with the total program memory space, for all the benchmarks we tested.

Although memory leaks are not the major cause of memory overhead in our experiments, they may be a serious problem with long-running programs like server applications where leaks can stack up indefinitely over a long period of program execution. One promising way to cope with the problem is to integrate to CRCount a garbage collection mechanism for reclaiming the leaked memory. Whenever the amount of memory occupied by the undeleted objects exceeds certain limit, we can scan the entire memory of a program and mark all the objects that are referred to by pointers. At this time, all the undeleted objects that have not been marked while scanning the memory obviously correspond to the memory leaks. Now we can reclaim the memory occupied by the identified memory leaks by releasing forcibly. Since CRCount already has a bitmap that pinpoints the pointers from the vast program memory, the garbage collection can be performed more efficiently and accurately than conservative garbage collectors. We have implemented a simple garbage collector to measure how much performance overhead it incurs. The garbage collection starts from the pointers in the stack, the global variables, and the registers, and follows the pointers recursively to scan the pointers in the heap region. All the objects referred to by the pointers are marked (using the reserved field in the per-object metadata) and all the memory leaks are released at the end. We ran `gcc` with the garbage collector enabled because it shows the largest amount of memory occupied by the undeleted objects and thus is expected to give us the worst case performance overhead among the benchmarks. We used three different threshold values (64MB, 128MB, and 256MB) and let the garbage collector run whenever the amount of memory occupied by the undeleted objects exceeded these values. Compared to the version without the garbage collector, it showed an overhead of 2.3%, 1.1%, 0.4%, respectively. We believe that this overhead is acceptable to be integrated into CRCount.

VIII. Security Analysis

In this section, we perform the security evaluation by running CRCount-enabled programs with real vulnerability exploits. We also discuss some of the security considerations for CRCount.

A. Attack Prevention

To evaluate the effectiveness of CRCount in mitigating UAF errors, we ran several applications with publicly available vulnerability exploits. Table III shows the list of vulnerabilities tested with CRCount. CRCount successfully detected the double free and invalid free vulnerabilities. We explain the test results with the UAF exploits below.

All the UAF exploits we used accessed the freed region only before it is reallocated. Thus, the UAF accesses in the exploits did not affect the original build of the target program. Note that CRCount is purposed to prevent the attackers from reallocating an object in the memory region still pointed to by the dangling pointers; thus, it did not affect the tested exploits. However, in order for these exploits to eventually be developed into serious attacks, the freed region should be reallocated so that the UAF access can read from/write to the allocated victim object. If CRCount correctly keeps track of the reference counts in the tested programs, it will properly mitigate these advanced exploits. We will show that it is indeed the case in a moment.

For CVE 2016-6290, CRCount detected a double free vulnerability while the original build of the program did not. We found that the double free was triggered by a pointer that still referred to a freed object. The original build of the program did not detect it because another object was allocated at the same address before the free function is called with the dangling pointer. This shows that CRCount successfully delayed the freeing of the object with pointers still referring to it.

To verify that CRCount properly delays the reuse of problematic memory region in the exploits, we have also implemented an extended version of CRCount with a UAF detection capability, called *CRCount-det*. CRCount-det performs checks on every memory access to see if the accessed heap object is marked as freeable. While extra checks on memory accesses cause non-trivial performance overhead, we would immediately know if a pointer is used to access an undeleted object. In our experiments, CRCount-det could detect all the UAF attempts we tested, which also implies that CRCount would properly delay freeing of the object to prevent malicious attempts utilizing the tested vulnerabilities.

B. Security considerations

One of the concerns about the security guarantee of CRCount is how effective a delayed-memory-reuse based mitigation is against UAF exploits. Recall that one key condition in exploiting an UAF exploit is to locate an attacker-controlled object into the freed memory region pointed to by dangling pointers in order to arbitrarily control the results caused by dangling pointer dereferences. However, in a victim process that CRCount is applied, when an object is freed, no objects are allocated until the reference count becomes zero. At this point, the objects can be accessed only through the existing links (pointers), maintaining their original semantics. Namely, the attacker can no more implant any controllable object into the freed memory region where dangling pointers still point to. As a result, the attackers' capabilities are limited to performing the actions that are originally allowed for the object in the program, unless the attackers use another kind of vulnerability. This makes it impossible, or makes it significantly complicated at least, for the attackers to achieve their goal. It is noteworthy that CRCount nullifies any heap pointers inside the object when the object is freed, so the attackers are further restricted from reusing the heap pointer inside the object.

IX. Limitations

Custom Memory Allocator. While applying CRCount to the benchmark programs, we encountered some cases (i.e., `gcc` in SPEC CPU2006 and `freqmine` in PARSEC) where the program had to be patched in order for our technique to work correctly. Specifically, the problem occurred mainly due to the use of a custom memory allocator that internally allocates objects from a reserved chunk of memory without going through the expensive heap management functions. If different types of objects are allocated to the same memory region, the pointers that were stored in the previous object can be overwritten by a non-pointer-type value in the newly allocated object. Had CRCount been able to identify the custom deallocator paired with the custom allocator, it would insert a runtime library call to handle the pointers enclosed in a freed region. Since it was not, we needed to manually identify these custom memory deallocators and explicitly insert the CRCount's runtime library calls to update the pointer bitmap and the reference counts. Specifically, we added 2 lines to `gcc` and 1 line to freqmine to call `crc_free` upon custom memory deallocation.

Unaligned Pointer. Another problem we met in the experiments is that some of the programs stored pointers in 4-byte aligned addresses, which is finer than the assumed alignment (i.e. 8-byte) in the pointer bitmap. Specifically, PARSEC's `freqmine` benchmark used a custom allocator that aligns objects at a 4-byte boundary. We addressed this by modifying the custom memory allocator to align objects at a 8-byte boundary. Also, `Apache` web server used `epoll_event` struct defined with `__attribute__((packed))`, which made the pointer inside the struct to be located at a 4-byte boundary. We addressed this by wrapping the struct so that the pointer is located with an 8-byte alignment. Note that CRCount could just ignore the unaligned pointer store by not increasing the reference count for the stored pointer. We chose to patch the code for more complete protection. 12 lines were modified in `freqmine` and 10 lines in `apache` to ensure pointers are stored at aligned addresses.

Vectorization Support. Our prototype CRCount implementation currently does not support vectorization in LLVM IR. DangSan also does not support vectorization—it simply ignores the stores of vector types. Even though vector operations rarely have to do with pointer values, as ignoring the vector types could adversely affect reference counter management, we instead turned off vectorization in all the experiments. It is our future work to correctly deal with the vector types in our analysis and instrumentations.

Limitations of Pointer Footprinting. There are cases where our static analysis fails to determine whether a particular store instruction should be instrumented or not. We perform only intra-procedural backward data flow analysis. Thus, if a pointer is cast before being passed to a function, we cannot analyze how the pointer is cast, and thus we may fail to correctly decide whether to instrument the store instruction or not. However, since we used LLVM link-time optimization (LTO), many functions are inlined to their caller, which enabled us to get much information from the backward data flow analysis. Another problem regarding static analysis is that we cannot track type unsafe pointer propagation through memory. For example, a pointer could be cast to an integer, stored in some integer field of a struct type variable, and passed around the program through memory as an integer. The pointers stored as an integer data in this process will not increase the reference counts of their corresponding objects. This is a common limitation faced by every approach based on pointer tracking [36], [17], [40], [23], [24]. Like all the approaches based on source code, we cannot instrument the libraries distributed as a binary file. This can cause errors in reference counter management if a pointer stored in the instrumented program is killed in such uninstrumented binary libraries.

X. Conclusion

CRCount is our novel solution to cope with UAF errors in legacy C/C++. For efficiency, CRCount employs the implicit pointer invalidation scheme that avoids the runtime overhead for explicit invalidation of dangling pointers by delaying the freeing of an object until its reference count naturally reduces to zero during program execution. The accuracy of reference counting greatly influences the effectiveness of CRCount. Therefore in our work, we have developed the pointer footprinting technique that helps CRCount to precisely track down the location of every heap pointer along the execution paths in the legacy C/C++ code with abusive uses of type unsafe operations. CRCount is effective and efficient in handling UAF errors in legacy C/C++. It incurs 22% performance overhead and 18% memory overhead on SPEC CPU2006 while attaining virtually the same security guarantee as other pointer invalidation solutions. In particular, CRCount has been more effective for programs heavily using pointers than other solutions. We claim that this is an important merit because UAF vulnerabilities are more likely prevalent in those programs.

Acknowledgment

The authors would like to thank Lucas Davi for shepherding the paper. This work was partly supported by MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2018-2015-0-00403) supervised by the IITP(Institute for Information & communications Technology Promotion), the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (NRF-2017R1A2A1A17069478, NRF-2018R1D1A1B07049870), IITP grant funded by the Korea government(MSIT) (No.2016-0-00078, Cloud based Security Intelligence Technology Development for the Customized Security Service Provisioning; No.2018-0-00230, Development on Autonomous Trust Enhancement Technology of IoT Device and Study on Adaptive IoT Security Open Architecture based on Global Standardization [TrusThingz Project]). The ICT at Seoul National University provides research facilities for this study.

References

[1] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers." in *USENIX Security Symposium*, 2010, pp. 177–192.

[2] A. Alexandrescu, *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

[3] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages," in *Acm sigplan notices*, vol. 41, no. 6. ACM, 2006, pp. 158–168.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[5] H. Boehm, A. Demers, and M. Weiser, "A garbage collector for c and c++," 2002.

[6] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 133–143.

[7] T. H. Dang, P. Maniatis, and D. Wagner, "Oscar: A practical page-permissions-based scheme for thwarting dangling pointers," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 815–832.

[8] M. Daniel, J. Honoroff, and C. Miller, "Engineering heap overflow exploits with javascript." *WOOT*, vol. 8, pp. 1–6, 2008.

[9] D. Gay, R. Ennals, and E. Brewer, "Safe manual memory management," in *Proceedings of the 6th international symposium on Memory management*. ACM, 2007, pp. 2–14.

[10] S. Ghemawat and P. Menage, "Tcmalloc: Thread-caching malloc, 2007," *URL {http://goog-perftools. sourceforge. net/doc/tcmalloc. html}*, 2005.

[11] I. Haller, E. Van Der Kouwe, C. Giuffrida, and H. Bos, "Metalloc: Efficient and comprehensive metadata management for software security hardening," in *Proceedings of the 9th European Workshop on System Security*. ACM, 2016, p. 5.

[12] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[13] M. Hertz and E. D. Berger, "Quantifying the performance of garbage collection vs. explicit memory management," in *ACM SIGPLAN Notices*, vol. 40, no. 10. ACM, 2005, pp. 313–326.

[14] M. Hirzel and A. Diwan, "On the type accuracy of garbage collection," *ACM SIGPLAN Notices*, vol. 36, no. 1, pp. 1–11, 2001.

[15] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 437–452.

[16] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2014, pp. 276–291.

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

[18] K. Lee, "Memory management," in *Pro Objective-C*. Springer, 2013, pp. 53–74.

[19] A. Mazzinghi, R. Sohan, and R. N. Watson, "Pointer provenance in a capability architecture," in *10th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 2018)*, 2018.

[20] S. Nagaraju, C. Craioveanu, E. Florio, and M. Miller, "Software vulnerability exploitation trends," *Microsoft Corporation*, 2013.

[21] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3. IEEE Computer Society, 2012, pp. 189–200.

[22] ——, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2014, p. 175.

[23] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[24] ——, "Cets: compiler enforced temporal safety for c," in *ACM Sigplan Notices*, vol. 45, no. 8. ACM, 2010, pp. 31–40.

[25] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proceedings of the 3rd international conference on Virtual execution environments*. ACM, 2007, pp. 65–74.

[26] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software." in *NDSS*, vol. 5. Citeseer, 2005, pp. 3–4.

[27] G. V. Nishanov and S. Schupp, "Garbage collection in generic libraries," *ACM SIGPLAN Notices*, vol. 34, no. 3, pp. 86–96, 1999.

[28] G. Novark and E. D. Berger, "Dieharder: securing the heap," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 573–584.

[29] M. Olsson, "Smart pointers," in *C++ 17 Quick Syntax Reference*. Springer, 2018, pp. 157–160.

[30] J. Rafkind, A. Wick, J. Regehr, and M. Flatt, "Precise garbage collection for c," in *Proceedings of the 2009 international symposium on Memory management*. ACM, 2009, pp. 39–48.

[31] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker." in *USENIX Annual Technical Conference*, 2012, pp. 309–318.

[32] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, "Freeguard: A faster secure heap allocator," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2389–2403.

[33] M. S. Simpson and R. K. Barua, "Memsafe: ensuring the spatial and temporal memory safety of c at runtime," *Software: Practice and Experience*, vol. 43, no. 1, pp. 93–128, 2013.

[34] P. Sobalvarro, "A lifetime-based garbage collector for lisp systems on general-purpose computers." MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, Tech. Rep., 1988.

[35] A. Sotirov, "Heap feng shui in javascript," *Black Hat Europe*, 2007.

[36] E. van der Kouwe, V. Nigade, and C. Giuffrida, "Dangsan: Scalable use-after-free detection," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 405–419.

[37] P. R. Wilson, "Uniprocessor garbage collection techniques," in *Memory Management*. Springer, 1992, pp. 1–42.

[38] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 414–425.

[39] T. Yamauchi and Y. Ikegami, "Heaprevolver: Delaying and randomizing timing of release of freed memory area to prevent use-after-free attacks," in *International Conference on Network and System Security*. Springer, 2016, pp. 219–234.

[40] Y. Younan, "Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers." in *NDSS*, 2015.