Sound Garbage Collection for C using Pointer Provenance

SUBARNO BANERJEE, University of Michigan, USA
DAVID DEVECSERY, Georgia Institute of Technology, USA
PETER M. CHEN, University of Michigan, USA
SATISH NARAYANASAMY, University of Michigan, USA

Garbage collection (GC) support for unmanaged languages can reduce programming burden in reasoning about liveness of dynamic objects. It also avoids temporal memory safety violations and memory leaks. *Sound* GC for weakly-typed languages such as C/C++, however, remains an unsolved problem. Current value-based GC solutions examine values of memory locations to discover the pointers, and the objects they point to. The approach is inherently unsound in the presence of arbitrary type casts and pointer manipulations, which are legal in C/C++. Such language features are regularly used, especially in low-level systems code.

In this paper, we propose Dynamic Pointer Provenance Tracking to realize sound GC. We observe that pointers cannot be created out-of-thin-air, and they must have provenance to at least one valid allocation. Therefore, by tracking pointer provenance from the source (e.g., `malloc`) through both explicit data-flow and implicit control-flow, our GC has sound and precise information to compute the set of all reachable objects at any program state. We discuss several static analysis optimizations, that can be employed during compilation aided with profiling, to significantly reduce the overhead of dynamic provenance tracking from nearly 8× to 16% for well-behaved programs that adhere to the C standards. Pointer provenance based sound GC invocation is also 13% faster and reclaims 6% more memory on average, compared to an unsound value-based GC.

CCS Concepts: • **Software and its engineering → Garbage collection**; *Automated static analysis.*

Additional Key Words and Phrases: Garbage Collector Safety, Pointer Provenance, Optimistic Hybrid Analysis

ACM Reference Format:
Subarno Banerjee, David Devecsery, Peter M. Chen, and Satish Narayanasamy. 2020. Sound Garbage Collection for C using Pointer Provenance. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 176 (November 2020), 28 pages. https://doi.org/10.1145/3428244

1 INTRODUCTION

Unmanaged languages such as C/C++ are the languages of choice for a vast set of large, complex, ubiquitous, and critical software bases, such as Linux, openssl, MySQL, and these languages continue to be popular among many developers. Unmanaged languages require programmers to explicitly allocate and free memory space. This requirement not only increases programming burden, but is also a source of common classes of bugs: use-after-free and memory leaks. Use-after-free bugs are not just a reliability issue, but a significant source of security vulnerabilities in modern systems [Caballero et al. 2012], as they compromise temporal memory safety [Nagarakatte et al. 2010]. In spite of significant advancements, prior solutions for temporal memory safety incur prohibitive performance overheads (~ 60% [Nagarakatte et al. 2010; Simpson and Barua 2013;

Authors' addresses: Subarno Banerjee, University of Michigan, USA, subarno@umich.edu; David Devecsery, Georgia Institute of Technology, USA, ddevec@gatech.edu; Peter M. Chen, University of Michigan, USA, pmchen@umich.edu; Satish Narayanasamy, University of Michigan, USA, nsatish@umich.edu.

2475-1421/2020/11-ART176
https://doi.org/10.1145/3428244

Zhang et al. 2019]). Memory leaks also compromise system reliability and can cause unpredictable performance [Vilk and Berger 2018]. Prior solutions have tried to address memory leaks through a combination of offline bug detectors [Caballero et al. 2012; Hastings and Joyce 1991], and runtime systems that probabilistically repair these bugs [Berger and Zorn 2006; Dhurjati and Adve 2006; Dhurjati et al. 2003].

Replacing manual deallocation of memory in unmanaged languages with a sound and efficient garbage collector (GC) would address all of the above problems by guaranteeing temporal memory safety, avoiding memory leaks, and reducing programmer burden. Unfortunately, a sound GC for weakly-typed languages like C/C++ has remained elusive.

A sound GC is one that guarantees to not free an object that is accessed later [1]. Typically, a sound mark-and-sweep GC [Cohen 1981] automatically reclaims memory at runtime by freeing a set of objects that can be guaranteed to be unreachable from a set of "root" pointers (pointers in global variables, stack variables, and registers). To compute this set at runtime, given a pointer, a GC should be able to (1) identify a pointer's *dynamic points-to* object, that is, the object reached by dereferencing the pointer, and (2) locate all the pointers contained in that reachable object. We refer to the latter set as the *pointers-within* set for an object. In strongly-typed, memory-safe languages like Java, both of these operations are straightforward [Bacon et al. 2001]. A pointer's value can be used to identify its dynamic points-to object due to spatial memory safety [Nagarakatte et al. 2009], and the pointers-within set of an object can be easily determined due to the strong type system.

C/C++, however, is weakly-typed. Pointer values can reside in, or be computed from, non-pointer variables, making it difficult to locate them within a reachable object at runtime. Even if we can locate all the locations with pointer values, they are not guaranteed to point within the referenced objects. This is true even in spatially memory safe programs, as a pointer value may be arbitrarily transformed to point away from the object, then manipulated back just before a dereference. Such pointer manipulations are regularly used in low-level systems code [Memarian et al. 2019].

Prior works on GC for C/C++ [Boehm 1993, 2004] have tried to overcome some of these problems using *value-based* heuristics, but these works do not guarantee soundness. They assume that only memory locations with values within an allocated heap object's address range are valid pointers, and that the value points within the referent object. This assumption is *unsound* as they cannot identify the referent object when a pointer value goes out-of-bounds due to arbitrary pointer manipulations allowed in C/C++. They are also *imprecise*, and therefore, prone to memory leaks when non-pointer locations hold values that happen to be within the heap address range. Finally, they have to examine the value of every reachable memory location to determine if it is a pointer or not, leading to higher performance overhead.

In this paper, we design the first *sound* GC for C/C++ called *Provenance-based Garbage Collection* (Prov-GC). We observe that a C/C++ program cannot create a pointer out-of-thin-air; instead, pointer values must be derived from a *valid pointer source*. Valid pointer sources are from *allocation functions* (e.g. `malloc`) and the *address-of* (`&`) operator. These pointer values subsequently propagate to other variables either through explicit data-flow or implicit control-flow. Thus, our key idea is to use dynamic information-flow tracking to soundly and precisely determine the set of all memory locations that hold values derived from pointers, and the object locations they point-to. Our mark-and-sweep GC uses this information to soundly reclaim unreachable objects.

Conventional dynamic information-flow tracking (DIFT), however, is known to incur significant performance overhead, slowing execution down by several times [Clause et al. 2007]. This is due to the need to execute a "monitor" typically for every instruction that could propagate a

[1] Ideally GCs would free objects that *will not* be accessed later, but real GCs settle for the *cannot* be accessed approximation.

"taint". Furthermore, taint propagation through implicit control-flow (a necessity for us to ensure soundness) is known to be not only expensive in terms of performance, but also can imprecisely taint a significant fraction of memory locations [Clause et al. 2007].

We observe several optimization opportunities that we exploit using hybrid taint analysis [Banerjee et al. 2019] to realize a low-overhead DIFT for pointer provenance. A common property that our static analysis exploits is that if an instruction destination operand's taint meta-data is guaranteed to not change, then the taint monitor for that instruction can be elided. This property holds true for instructions that are non-pointer operations (e.g., `inta = intb * 10;`), which is the vast majority of the instructions executed. We also show that if an instruction's destination operand is a statically declared pointer and its source operand is guaranteed to be a "safe" pointer (value points within the object), then its taint value is known at compile time, and therefore a runtime monitor is unnecessary (e.g., `int* ptr = safe_ptr;`). This category includes common pointer assignments, where the pointer has not been manipulated arithmetically. Finally, if the source and destination operands of an instruction are the same (e.g., `ptr = ptr+4;`), then there is no need to update the destination's taints.

Somewhat surprisingly, tracking implicit information flows, known to be intractable in general, turns out to be practical for pointer provenance. Conditional branches dependent on pointer variables can propagate taint implicitly to its control-dependent instructions. While we do not expect reasonable programmers to use such programming constructs, to realize sound GC, we must consider its possibility, as they are legal in C/C++. Fortunately, we are able to show that when branch conditions are based on comparisons between in-bounds pointers or with `NULL` value, there is insufficient implicit information flow to require dynamic tracking. While we need modest dynamic checks to establish that pointer values are in-bounds, we never have to propagate implicit flow for the programs we studied.

Finally, to prove the above properties statically to aggressively elide dynamic taint monitors, we apply Optimistic Hybrid Analysis (OHA) [Banerjee et al. 2019; Devecsery et al. 2018]. OHA uses profiled likely invariants to predicate the whole-program context-sensitive flow-sensitive static taint analysis.

The C standard [ISO 2018] specifies several restrictions on using pointers and operations that can be performed on pointer types. We show that assuming these properties benefits our pointer provenance tracking significantly, and reduces the overall cost of GC for standard-compliant programs. We provide solutions with and without this optimization, because, in practice, there are many programs that regularly violate the standard [Memarian et al. 2016], and as such they require provenance tracking without optimizations that depend on these C standard specifications.

We evaluate our Prov-GC tool on several long-running large applications as well as memory-intensive benchmark programs. Unlike the Boehm-Demers-Weiser GC (BDW-GC) [Boehm 2004], Prov-GC is sound. For well-behaved programs that are C standards-compliant, we pay only an additional 16% average performance cost to dynamically track pointer provenance. Of which, 14% is due to explicit data-flow tracking, and the remaining is to track implicit-flow. We find that our optimistic optimizations that elide dynamic monitors are very effective with adequate profiling. Without them, we see nearly 8× slowdown. In addition to soundness, compared to BDW-GC, the performance of our GC invocation is about 13% faster, as we avoid scanning, and reclaim about 6% more memory per GC invocation due to our GC's improved precision.

16% performance overhead of our GC is especially appealing as it obviates the need for a slower dynamic temporal safety solution (60% overhead [Zhang et al. 2019]), besides reducing programming burden and avoiding memory leaks.

We make the following contributions in this paper:

- We present Prov-GC, a GC that is sound for all legal C/C++ programs. Previous GC solutions for C/C++ are unsound as they might free reachable objects.
- We present the idea of dynamic pointer provenance, and use it to realize a sound GC for C/C++.
- We show how we can elide taint monitors for a vast majority of instructions such as operations on non-pointers, "safe" pointers, etc.
- We show tracking implicit information-flow in the context of pointer provenance is necessary and practically feasible.
- We show how the C standard specifications induce a significantly improved provenance tracking solution for standard-compliant programs.
- We apply optimistic hybrid analysis [Devecsery et al. 2018] to optimize dynamic pointer provenance and realize an efficient GC that incurs 16% overhead. This overhead is much lower than dynamic temporal safety checking, and it avoids memory leaks and reduces programmer burden.

2 BACKGROUND AND MOTIVATION

This section discusses the motivation for using GC in weakly-typed languages like C/C++, and the unsolved problems in realizing a sound GC for them.

2.1 Why GC for C/C++?

GC obviates the need for manual memory management and thereby eliminates two common classes of bugs in unmanaged languages: memory leaks and use-after-free [Caballero et al. 2012; Vilk and Berger 2018].

Memory leaks and use-after-free bugs are considered important classes of bugs and thus have received significant attention from academia and industry, who have tried to address these bugs through a variety of methods. For memory leaks, there exists a number of offline debugging tools and runtime probabilistic methods to mitigate the ill effects of these bugs [Bond and McKinley 2006; Clause and Orso 2010; Hauswirth and Chilimbi 2004; Jump and McKinley 2007; Novark et al. 2009; Rudafshani and Ward 2017; Vilk and Berger 2018; Xu et al. 2011]. We argue that GC for C/C++ would address the memory leak problem more comprehensively than these methods.

Use-after-free bugs are particularly important because they compromise system security. These bugs violate temporal memory safety [Nagarakatte et al. 2010], which along with spatial safety is necessary to ensure full memory safety. Recognizing the importance of memory safety, even commodity processors (e.g., Intel MPX) [Oleksenko et al. 2018] have started providing specialized hardware support for efficiently implementing spatial memory safety checks. Spatial memory safety, however, solves only part of the problem. Efficiently guaranteeing temporal safety remains expensive, as state-of-the-art solutions incur $\sim 60\%$ performance overheads [Zhang et al. 2019].

We argue that if we can realize a sound and efficient GC for C/C++, it would not only reduce the burden on future software development, but also help improve reliability and security of both future *and* legacy systems. If the execution time overhead of GC can be made lower than the overhead of other temporal safety solutions, then it certainly would be a superior solution, as it not only removes temporal errors, but also improves programmability.

2.2 GC and its Pointer Data Requirements

Identifying dead objects precisely at a given instant of a program's execution is hard as it depends on future execution. Therefore, current GCs conservatively identify live objects by assuming that the set of all "reachable" objects from a "root" set of pointers are live. The root-set consists of

all pointers in the registers, global and stack address space. A reachable object can still be dead as it may never get referenced in future. Current GCs use either incremental mark-and-sweep [Boehm 2004] or reference counting [Shahriyar et al. 2012, 2013] to compute reachability. In this paper, we use mark-and-sweep, though our provenance-based approach could also be used by a reference-counting GC.

When a mark-sweep GC is invoked, it performs two separate steps: *marking* computes the live set of objects that are transitively reachable from a *root set* of pointers, and *sweeping* reclaims memory from unreachable objects. To perform the reachability analysis, the mark step requires two crucial pieces of information about pointers: a pointer's **points-to object** (PT), and an object's **pointers-within set** (PW). The points-to object of a pointer is the dynamic object that can be dereferenced using that pointer. The pointers-within set of an object is the set of all pointers that are contained within that object.

Both points-to and pointers-within data are straightforward to determine *soundly* and *precisely* in type and memory safe languages such as Java, Python, and C#. The points-to object of a pointer can be determined from its value due to spatial memory safety. That is, a pointer's value is guaranteed to be within the address range of its points-to object. Furthermore, due to type safety, given an object, it is possible to precisely and quickly determine its pointers-within set, because only variables that are typed as pointers can hold pointer data; non-pointer variables cannot hold pointer data. *Determining points-to and pointers-within data in weakly-typed languages like C/C++, however, is a significant challenge as we discuss next.*

2.3 Value-Based GCs for C/C++ are Unsound

All prior attempts to provide GC for C/C++ use *value-based* heuristics to compute points-to and pointers-within information. For example, the best-known such work [Boehm 2004] computes the pointers-within set of an object by scanning every pointer-sized field in the object and checking if its value falls within the address range of any allocated heap object. If the check succeeds, then that field is assumed to be a pointer, and the points-to object for that field is assumed to be the allocated heap object whose address range includes that field's value.

Value-based GCs work by assuming that any allocated heap object that may be referenced in the future has at least one live register or memory location pointing to it at all times. This assumption may be violated, for instance, if the C program breaks any of the following three assumptions: (A1) Pointers are only stored in variables that are declared to be pointers or in sufficiently large integral type that can hold a pointer. (A2) If a memory location's value falls within the bounds of an allocated heap object then it *is* a valid pointer, or else it is a non-pointer. (A3) A pointer discovered through its value is assumed to point within its points-to object. Value-based GCs are **unsound** whenever any of the above invariants is violated.

In C/C++, even legal programs can violate these three assumptions because C/C++ has a weak type system and allows programs to store pointer values in integers and manipulate them in arbitrary ways. Figure 1 shows an example of a XOR linked list, a clever representation of a doubly linked list used in memory constrained embedded systems [xor 2004]. Each node stores the XOR of pointers in the two directions and recovers them using XOR operations during traversal. Note that none of the inner nodes store literal values of pointers, but have information encoded to reconstruct two valid pointers. A purely value-based GC approach can incorrectly reclaim objects pointed to by such *hidden* pointers, e.g. node C's address is not stored literally anywhere in the program state. This is a legal C program which can break the correctness of a value-based GC. More extreme examples are also possible via casting and other manipulations; for example, a program may split a

```
1  typedef struct { uintptr_t val, xptr; } xorlist;
2  xorlist *head = NULL, *tail = NULL;
3  void traverse(xorlist *start) {
4    xorlist *prev = NULL, *curr = start;
5    while (curr) {
6      printf("%ld\n", curr->val);
7      uintptr_t next = (uintptr_t) prev
                      ^ curr->xptr;
8      prev = curr;
9      curr = next;
10   }
11 }
```

```
12 void main() {
13   insert(...);
     ...
14   traverse(head); // use xptr to traverse
15   traverse(tail); // in both directions
16 }
```

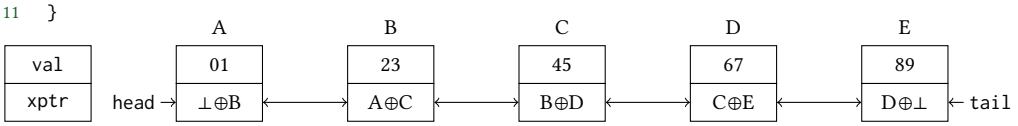

Fig. 1. Doubly Linked Lists can save space by storing the XOR of previous and next node pointers in a single integer location; `uintptr_t` is sufficiently long to hold pointer values. The inner nodes never store literal pointer values, but have sufficient information to reconstruct valid pointers to its two adjacent nodes.

pointer into several smaller integers then reconstruct the pointer later; this would violate all three assumptions.

Value-based GCs can also be **imprecise** because they may think a non-pointer is a pointer when its value happens to lie within the heap address range. If this non-pointer's value points-to an unreachable object, then GC would avoid reclaiming it. This can lead to memory leaks and lower performance. There has been follow-up work [Henderson 2002; Rafkind et al. 2009] that addressed this problem by adding another assumption that pointers only reside in declared pointer typed variables. This approach achieves greater precision, but sacrifices even more soundness, as it would ignore an integer value derived from a pointer through a cast.

Value-based GCs can also incur significant overhead while scanning the state space for pointers. Given a reachable object, GC has to scan each of its fields, and check if it could be a valid pointer or not. The check involves looking into a data-structure that maintains the address ranges of all heap objects.

In this work, we improve upon pure value-based GCs by using dynamic pointer provenance to soundly and precisely determine the set of all pointers and the objects they point-to. Also, this can quickly identify pointers-within set of an object without scanning each field, improving GC performance.

2.4 Need for Sound GCs

Guaranteeing correct GC behavior– i.e. objects reachable from the root set of pointers will not be freed, is important for *all* programs in the same way that it is important to have a sound compiler or runtime. Value-based GCs impose additional restrictions, as seen in §2.3 earlier, making them work correctly only on a subset of the language. These language properties might not be followed by legacy programs, and can be generally difficult to verify for new programs and compiler implementations that strive to conform to the standards but still may not. Moreover, new programs also reuse existing library code. So, there is value in supporting sound GC behavior for all programs without imposing additional restrictions on the language itself.

3 PROVENANCE-BASED GARBAGE COLLECTION

In this section we discuss the design of a sound and efficient provenance-based garbage collector. We will first motivate how a provenance-based garbage collector solves the soundness issues of prior GCs. Then we outline a simple-but-sound strawman GC. Finally, we discuss optimizations that reduce the overhead of our strawman GC, leading to a provenance-based GC that is both fast and sound.

We assume that, aside from temporal safety errors, the given program is a valid C/C++ program, and that it obeys the properties necessary for the compiler and hardware to guarantee a well-defined behavior. This includes spatial memory safety [Nagarakatte et al. 2009] and data-race-freedom [Adve and Hill 1990], which many prior works have addressed.

3.1 The Soundness of Provenance-Based GC

In this work, we argue that instead of using the *value* of a pointer to identify its points-to set, a GC can use the *provenance* of the pointer to soundly derive its dynamic points-to.

The soundness of our provenance-based solution is based on the assumption that allocated heap memory addresses cannot appear *out-of-thin-air*. That is, without knowing the return value of a call to an allocation function (e.g. `malloc` in C), it is impossible for the programmer to compute the address of any given dynamically allocated heap object. This assumption is true of most real type-unsafe languages, such as C. Given this assumption, any well-behaved program must ensure that any heap addresses dereferenced by a load or store operation are derived from the return of heap allocation functions. Consequently, an object allocated within the heap is only reachable in the future if it has one or more live register or memory values (henceforth *values*) which draw provenance from its allocation function's return value.

Throughout the remainder of this paper, we will use the term *pointer* to refer to a register or memory location whose value is directly or indirectly derived from one or more allocation return values. We use the term *points-to set* of a location to refer to the set of object allocation function return values from which the pointer is derived. The points-to set may be thought of as the set of objects that this pointer may be used to access in the future. A memory location with empty points-to set is not a pointer.

Since a heap pointer value cannot appear *out-of-thin-air*, and all pointers must have a provenance to at least one valid allocation, it is therefore *sufficient* to track all points-to sets for all pointers to reconstruct all currently reachable objects.

3.2 A Simple Provenance-Based GC

To show how pointer provenance can construct a sound GC, we present a simple, strawman design of a provenance-based GC.

Our strawman GC will naively track the points-to set for every register or memory location (henceforth *location*) in an execution by constructing a map from each location to the set of heap object allocations its value is derived from.

To dynamically track the points-to sets of all pointers, we apply a standard dynamic information flow (DIFT) policy, treating all heap object allocations as sources and using both data-flow and implicit-flow taint tracking. More specifically, the program begins with each location's points-to set empty. Whenever an allocation function returns, we add that allocation to the destination location's points-to set. Thereafter, whenever the program modifies a location, the points-to set of that location will be updated to contain the union of the points-to sets of any pointers it depends on. We note that for this analysis to be sound, when a location is modified, it must consider not

only data-flow dependencies, but also any implicit control-flow dependencies as well (e.g., when branch conditions depend on pointers).

A properly constructed DIFT analysis [Austin and Flanagan 2009; Schwartz et al. 2010] will, by construction, ensure that the points-to set of each pointer is conservative. That is, if the pointer could be used to dereference an object in the future, that object will be within that pointer's points-to set.

In order to avoid scanning memory to locate pointers, we also maintain a pointers-within set mapping for each allocated heap object. The pointers-within set will logically contain the memory location of every pointer within the allocated heap object. The pointers-within set for an object can be trivially maintained by initializing the set to empty when the object is allocated, then adding a pointer to the set whenever such a pointer is stored to a location within the object.

Once the collection phase of the GC actually begins, our GC only needs to iterate through the root set of the program (any locations statically reachable - globals, or reachable from any stack frames and registers), add these locations to a temporary set called the working set. Then, the GC will iterate through the working set and for each object in the working set identify all pointers with the object's pointers-within set. The GC then adds any objects in the points-to sets of those pointers to both the working set, and a set of live objects. This process iterates until the live set does not change. Any object not in the set of live objects at the conclusion of the algorithm may then be reclaimed.

While this straw-man solution provides a sound GC, tracking provenance metadata through all operations and through implicit flows will typically be very expensive. Fortunately, most instructions operate on non-pointers. Also, in the common case, pointers stay within object bounds, and do not propagate pointer data to other locations through control flow. We leverage these properties to significantly reduce the amount of provenance tracking required to construct a sound GC.

3.3 Optimizing Explicit Provenance

The strawman system described earlier requires dynamically inserting a monitor (to propagate points-to set) on nearly every instruction within the program. This would result in very high overheads [Clause et al. 2007]. However, we observe that for a significant fraction of instructions, its runtime monitors do not change their destination operands' points-to (taint). We use static analysis to elide these dynamic monitors without losing soundness. This section discusses three optimizations that elide such redundant monitors: 1) eliding non-pointer tracking, 2) eliding "safe-pointer" tracking, and 3) eliding monitors for pointers with the same operands.

3.3.1 Non-Pointer Tracking Elision. Within C programs, the vast majority of computation operates on data which is not logically derived from pointer values. If we can statically prove that a location within the program has an empty points-to set, then that location is a non-pointer, and the dynamic run-time system need not dynamically track the points-to set of that location. This detection can be accomplished by using static information flow analysis [Banerjee et al. 2019; Sabelfeld and Myers 2003] to compute a sound may points-to set, then eliding dynamic points-to set operations on locations with an empty may points-to set. For the example in Figure 2, we can trivially elide any provenance tracking for line 5 since neither of its operands have data-flow from any pointers, in fact they are constants.

3.3.2 Safe Pointer Tracking Elision. Next, we observe that the vast majority of pointers in C programs (1) have exactly one object in their points-to sets (singleton set) and (2) have a value within the allocated memory range of that object (in-bounds). We call these *safe pointers*.

```
1  void explicit_flow() {
2   unsigned int n = 10, o = 1000;
3   obj* A = malloc1(n*sizeof(obj));   // PT(A) = {malloc1}
4   char** B = malloc2(n*sizeof(obj));// PT(B) = {malloc2}
5   long z = o / n;      // elided by E1
6   char* p = A;         // elided by E2
7   long d = B - A;      // PT(d) = {malloc2, malloc1}
    ...
8   for (unsigned int i = 0; i < n * sizeof(obj); i++) {
9     char* q = p + d; // PT(q) = {malloc1, malloc2},    elided by E2 and line 10
10    *q = p;
11    p = p + 1;         // PT(p) = {malloc1}, elided by E3
12  }
13 }
```

Fig. 2. Explicit pointer provenance propagation

Safe pointers are handled correctly by value-based GC. Because the pointer value can be used to dereference only one object, and since the pointer value is in-bounds, we can use its value to determine its object. For the same reasons, we do not need to track the points-to set of any location in the program we know is a safe pointer, as we can identify its points-to set from its value at collection time.

To perform this optimization, we perform a static data-flow analysis to identify which instructions in a program must define safe pointers. While statically identifying safe pointers precisely in a program is hard, we construct a sound but imprecise data-flow analysis as follows. A pointer defined by the assignment from an allocation function is clearly safe. Assignment from a safe pointer is also safe. The result of any operation is safe, provided it satisfies two conditions: (1) the operation has only one pointer operand, and that operand is a safe pointer, and (2) the operation is guaranteed to not modify the pointer to point outside the bounds of the object it references (provably in-bounds).

Leveraging Dereferences. Our static safe-pointer identification methodology is conservative, and consequently will falsely identify many safe-pointers as non-safe. It is sound, but potentially introduces unnecessary dynamic checks. To help identify additional sources of safe pointers, we observe that any time an address is dereferenced, it must be an in-bounds pointer, otherwise the program would have undefined behavior (violating spatial memory safety). If we can additionally prove that the dereferenced pointer's points-to set is a singleton set, then we know the pointer is a safe pointer. To accomplish this, we construct another static taint analysis, with the goal of identifying singleton taint sets. To construct such an analysis, we observe that a points-to set can only propagate from a singleton-set to a non-singleton-set when it depends on multiple pointers. Therefore, a static analysis can determine pointers that must have singleton points-to sets by checking if its transitive dependency set contains no operations with multiple pointer dependencies. We leverage this must-have singleton-points-to sets analysis with our observation about dereferenced pointers being in-bounds to identify an additional source of safe-pointers: dereferenced pointers with singleton points-to sets.

3.3.3 Equivalent Points-to Propagation. Our third optimization exploits the fact that many pointer redefinitions do not change the points-to metadata, and therefore they can be elided. This is trivially true when the source and destination pointer operands are the same. For the example in Figure 2, the provenance (points-to) of p on line 11 cannot change. It is possible for arithmetic operation on a pointer to result in an out-of-bounds value. But in a well-defined (spatially memory safe) program,

it cannot be dereferenced before it is reverted back to be within bounds. We use a static data-flow analysis that elides the monitor for an instruction when it can be proven that the provenance of its destination operand is same as its source either directly or transitively through data-flow.

A related optimization is that, if the provenance of a location remains constant within a loop, our analysis hoists it out of the loop through a loop invariant code motion [Aho et al. 2006].

In summary, we can elide provenance tracking operations when –

E1 All source operands have empty provenance.
E2 The resultant pointer is *safe*.
E3 The resultant pointer is assigned to the same identifier as the source operand, directly or via temporaries.

3.4 Optimizing Implicit Provenance

Although rare, it is both possible and legal for weakly-typed programs to deconstruct and reconstruct pointers through implicit flow operations, as shown in Figure 3. Traditionally, implicit information flow DIFT is known to have severe limitations as the majority of locations can get tainted, and doing so, as proposed in our strawman solution would result in very poor heap object collection rates and slow provenance tracking performance. However, recall that the goal of a sound provenance-based GC is not to ensure that no taint is lost, as a security analysis would, but rather to ensure that a pointer cannot be reconstructed from any provenance data. In order for an address range to be reconstructed, there must be enough data about the pointer propagated implicitly to definitely reconstruct it. We observe that for many comparisons, the binary outcome of that comparison doesn't propagate enough information to reconstruct the pointer, even if the comparison were made many times.

We consider two specific comparison cases for a valid in-bounds pointer `ptr1`:

I1 == or != `NULL`
I2 == or != another valid in-bounds pointer `ptr2`

The outcome of the comparison determines the value of `ptr1` from two possible partitions – $\mathcal{S}_1 = \{\texttt{NULL}\}$ *or* $\{\texttt{ptr2}\}$ and $\mathcal{S}_2$ = the set of all other valid pointers. When in $\mathcal{S}_1 = \{\texttt{NULL}\}$, `ptr1` is an invalid pointer. When in $\mathcal{S}_1 = \{\texttt{ptr2}\}$, `ptr1`'s (or `ptr2`'s) value cannot be deduced from their equality alone but must be explicitly carried in `ptr1` (or `ptr2`). When in $\mathcal{S}_2$, we have eliminated only one possible value and still need sufficient information to determine `ptr1`'s value. Sufficient information to recover the pointer must then propagate either via at least one explicit data flow, or via a series of $2^{64} - 1$ equality comparisons, which are unreasonable to do in any practical amount of time. So, we can safely elide tracking implicit provenance propagation via these constrained comparisons. For all other comparisons, our sound GC propagates the provenance set through implicit operations.

```
1  long implicit_copy(long ptr) {
2    long hidden_ptr = 0;
3    for (int i = 0; i < sizeof(ptr) * 8; i++ {
4      long mask = 1 << i;
5      if (ptr & mask) {
6        hidden_ptr |= mask;  // set bit in hidden_ptr
7      }
8    }
9    return(hidden_ptr);
10 }
```

Fig. 3. Copying a pointer via implicit flow

The above condition for in-bounds pointers can be guaranteed statically for safe pointers but must be checked dynamically for unsafe pointers. Prior work on memory safety has enabled efficient spatial bounds checking [Nagarakatte et al. 2009; Simpson and Barua 2013], and checks required for pointer comparisons only incur a fraction of those costs. Note, we cannot assume that the pointers used in comparison are guaranteed to be in-bounds. They may be out-of-bounds, and later become in-bounds before being dereferenced.

```
1  obj* A = malloc1(n*sizeof(obj));   // PT(A) = {malloc1}
2  char** B = malloc2(n*sizeof(obj));//PT(B) = {malloc2}
3  unsigned int o = 1000;
4  bool flag = false;
5  long x = A + o;    // PT(x) = {malloc1}
6  long y = B - o;    // PT(y) = {malloc2}
7  char* p = A;
8  if (B != NULL) flag = true; // elided by I1
   ...
9  if (A == p) flag = true;     // elided by I2
   ...
10 if (x == y) {
11   p = A + 2*o;     // PT(p) = PT(A) ∪ PT(x) ∪ PT(y)   = {malloc1, malloc2}
12 }
```

Fig. 4. Implicit pointer provenance propagation

Consider the example code in Figure 4. After the comparison on line 8, the `flag` being `true` simply indicates that the pointer `B` is non-`NULL` which cannot be used to recover a valid pointer within object *B*. Similarly after line 9, if `flag` is `true` (or `false`), you still need either (or both) of `A` and `p` to access the object(s). However, line 10 propagates sufficient information to recover a pointer value. When `x == y` succeeds, it encodes the distance between objects *A* and *B*, so that even if all pointers to *B* are discarded, a pointer to *B* can still be recovered as in line 11. To handle this information flow, we add the pointer provenance of line 10's comparison operands, `x` and `y`, to the control-dependent line 11's result `p`.

3.5 Other Points-To Set Propagation Channels

Pointer information can escape the managed address space and leak through external channels, such as by writing them to the file-system and reading them back. Safely handling such channels would require elaborate mechanisms to preserve the pointer provenance of such escaping values, and treat them as always live to exclude from being collected. Such pointer propagation channels being practically rare, we conservatively disable GC when any value escaping the program's address-space has non-empty pointer provenance.

3.6 C Standard for Pointers

The C standard [ISO 2018] places certain restrictions on the possible values of pointers, limiting acceptable pointer behaviors in correct programs with well defined behaviors on all platforms. The standard disallows arbitrary manipulations on pointers [§6.5.6], but allows arbitrary, implementation defined, conversions between integer and pointer types [§6.3.2.3]. As a result, pointer-typed values may be in one of three states: (1) **`in-bounds`**: well defined in-bounds values, (2) **`one-past-end`**: pointing to a location just past the end of an array, and (3) **`imp-def`**: an implementation-defined value converted from an integer, which is unknown in the general case. We will now show that Prov-GC can leverage these restrictions to expand the set of safe pointers (§3.3.2) to significantly reduce pointer provenance tracking.

If we ignore the `imp-def` case for now, then these properties are clearly highly advantageous to our garbage collector. Because, if all pointer typed values are in-bounds or one past the end of an array, then all pointer typed values are safe-pointers by definition. Thus, our GC can apply our optimization discussed in §3.3.2 for all pointer type values.

The `imp-def` case does not present an instance of an in-bounds pointer, as it allows an arbitrary value to be present in a pointer. Fortunately, however, the lack of definition between conversion from an integral to pointer-typed value disallows the program from reasoning about any value stored in that pointer, except under very specific conditions covered shortly. As any pointer in `imp-def` instance is not defined by the standard, a program cannot portably rely on it to reconstruct a pointer later, and thus it cannot be used to legally dereference or construct a pointer in the future, allowing Prov-GC to conservatively treat it as a safe-pointer.

The one exception we referred to is a defined conversion from a pointer value to an integral value and back as defined in [§7.20.1.4] of the C standard. This conversion applies to **`intptr_t`** values. For these values a **`void`** pointer may be converted to an **`intptr_t`** type and back. The conversion is not defined, except when the value stored in **`intptr_t`** variable is unchanged. Thus, any manipulated value of the integer would not have a standard-defined mapping when converted back to a pointer, and therefore our earlier conclusion for `imp-def` applies. If the value is unchanged when it is stored as **`intptr_t`**, then when it is converted back to a pointer type, it has to be either `in-bounds` or `one-past-end`.

Note that our example of XOR linked list in Figure 1 complies with the above restrictions because it only uses **`intptr_t`** type to convert pointers into integers, and it recovers the exact value of the original pointer using XOR operation before converting it back to a valid pointer.

We note that this optimization relies on the programmer writing strictly standard compliant portable C code. Many implementations of C compilers do define mappings when performing pointer to integer conversions, and many code-bases do legally (but not portably) rely on these facets of the compiler [Chisnall et al. 2015; Memarian et al. 2016]. As a result, we provide solutions with and without this optimization. Programs that strictly adhere to the C standard can take advantage of this optimization.

3.7 Optimistic Hybrid Analysis

Provenance-based GC relies heavily on dynamic taint tracking [Schwartz et al. 2010], and consequently can incur significant overheads. Fortunately, recent work has shown that dynamic taint tracking can significantly benefit from a technique known as optimistic hybrid analysis [Banerjee et al. 2019; Devecsery et al. 2018]. Optimistic hybrid analysis (OHA) is a method of dynamic analysis optimization based on the insight that optimization should be done for the common case. Traditional hybrid analyses use a sound static analysis to reason about *all possible* future executions (and many impossible ones, due to over-approximation). However, when optimizing a dynamic analysis, the optimization need only care about the execution that will actually happen. To help approximate this, OHA uses a *predicated* static analysis, which takes in a set of assumptions (called likely invariants), and only guarantees that the static analysis is sound for executions in which these likely invariants actually remain invariant. Assuming these invariants allows the static analysis to reason much more effectively about the analyzed program, dramatically improving its ability to reduce dynamic checks. In this case, it allows the dynamic taint analysis to be aggressively optimized by eliding taint tracking monitors along paths that do not propagate taints in the predicated static analysis. A runtime system then checks the likely invariants at runtime, and falls back to a conservative analysis if the invariants ever fail.

We leverage OHA to improve our provenance-based GC in two ways: (1) we use it to improve our static empty points-to analysis in §3.3, (2) we assume pointers used in comparisons are in-bounds, reducing the amount of implicit flow tracking done in §3.4. For our first use of OHA, we simply apply the same optimizations found in the Iodine tool [Banerjee et al. 2019] to improve our common-case identification of empty may-points-to sets. Our second use is slightly more subtle. A conservative analysis would require that we propagate implicit flow information for any pointer which may be out-of-bounds during the **I1** and **I2** implicit flow checks. However, it is *very* rare for a pointer used within a comparison to be out-of-bounds, so we assume the invariant that any pointer used in a comparison is in-bounds. Using this invariant, we can remove any implicit flow taint tracking that may occur in the common case, so long as we first dynamically verify that all pointers used in comparisons for branches are in-bounds. If the check fails before the comparison, we soundly switch to the conservatively optimized analysis that propagates taint through the implicit flow.

4 IMPLEMENTATION

In this section we discuss the implementation of Prov-GC, an instance of Provenance-based Garbage Collection for C programs. Prov-GC relies on three primary components: (1) a Static Pointer Provenance Analysis (PPA), (2) a Dynamic Pointer Provenance Tracking (PPT) instrumented on the target program, and (3) a Provenance-based GC library for use by the target program. Our static analysis, and dynamic analysis instrumentation is implemented in the LLVM 7.0 compiler infrastructure [Lattner and Adve 2004], and the Provenance-based GC library is a modification of the Boehm-Demers-Weiser GC [Boehm 2004] to use provenance metadata for GC. We discuss the implementation details below.

4.1 Static Pointer Provenance Analyses

The goal of our static analyses is to classify all LLVM static single assignment form [Alpern et al. 1988] values in a program into three partitions– non-pointers, safe-pointers, and unsafe pointers. Once this partition is computed, Prov-GC instruments instructions that define safe-pointers and unsafe-pointers with necessary PPT operations (only pointers-within set tracking for safe pointers, and both pointers-within and points-to tracking for unsafe), and no tracking is needed for non-pointers.

The first static PPA analysis we use is an information flow analysis that identifies values that *may* be influenced by pointer sources in two ways– a whole-program context-sensitive data-flow analysis, and a control-flow dependence analysis. To perform this, we construct a whole-program definition-use graph (DUG) [Aho et al. 2006]. We add a node to the DUG for an instruction that defines a value and edges to connect definitions to their uses. Our DUG is context-sensitive so that each function local DUG is replicated per call site, except for recursive calls, and expanded to create the inter-procedural DUG. This makes the information-flow analysis much more precise by distinguishing between distinct invocations of the same function. This DUG creation in turn requires a pointer-analysis to identify indirect information-flows via aliasing memory accesses, so as to add data-flow edges in the DUG from pointer definitions to its aliasing uses. To this end, we use Andersen's-based whole-program context-sensitive flow-insensitive inclusion-based pointer-analysis [oha]. Once the DUG is created, we assign an empty *static points-to* set to every value and initialize the set for values defined by pointer sources, i.e. a heap allocation function (e.g. `malloc`). It then traverses the whole-program DUG iteratively, accumulating the union of points-to sets of values that are used in a definition, until all points-to sets reach a fixed point. At the end of this data-flow analysis, all values with empty points-to sets are definitely non-pointers, and the rest are may-be pointers.

Optimistic Hybrid Analysis: Static data-flow analysis is traditionally imprecise (has many false positives), particularly inter-procedural pointer-analysis with indirect function calls becomes imprecise as well as unscalable. As a result many non-pointer values will be incorrectly identified as may-be pointers in a traditional version of our PPA static analysis. We tackle this imprecision problem using OHA [Devecsery et al. 2018], which assumes likely program invariants– such as unreachable code, indirect function callees, and unrealized call contexts, to predicate the DUG construction. As a result, the analysis becomes more scalable as well as precise, allowing us to discard many non-pointers and computing more precise points-to sets. We use Iodine [Banerjee et al. 2019], an OHA optimized taint analysis tool, with minimal changes to track the points-to sets as taint metadata.

[§3.3 **E1**] : OHA optimized data-flow analysis identifies non-pointers much precisely for which we can elide dynamic tracking.

Safe Pointer Optimizations [§3.3 **E2**] : So far, we've identified non-pointers; we further classify pointers into safe and unsafe sets as defined in §3.3.2. To do so, we construct a conservative data-flow analysis, identifying must-be safe-pointer operations as operations which may not pass through unsafe operations (where safe operations are those defined in §3.3.2). We find the following LLVM instructions to be safe: (1) assignments from values of equivalent size, (2) loads from addresses which have only had safe pointers stored to them, (3) LLVM GEP (pointer arithmetic) instructions in which the result is in-bounds of the allocation pointed to by the manipulated pointer, (4) passing as arguments in function calls, (5) returning pointers from function calls. We consider all other operations to be *unsafe*. Once we compute this analysis, we use it to elide points-to set tracking for any safe pointers.

Equivalent Points-to Optimizations [§3.3 **E3**] : The property reasons equivalence of a pointer's points-to set when a pointer value is assigned back to same source-level identifier as its source operand. To map SSA values to their source identifiers, we use LLVM's source-level debugging information, and check if values are assigned transitively to the same identifier. Then, we can elide dynamic tracking operations for all the definitions along the define-use chain. The metadata creation operation must still be performed for the first definition. When this first definition is inside a loop, we use the reaching definition from outside the loop that is used by this value to perform the points-to set tracking instrumentation, so that a later loop invariant code motion optimization can hoist this one-time metadata creation operation outside the loop. This optimization can benefit for example when traversing an array inside of a loop.

Implicit Flow Optimizations [§3.4 **I1**,**I2**] : Finally, the static analysis optimizes implicit provenance tracking. Since statically proving pointer values derived using arbitrary arithmetic *must-be* in-bounds is challenging, we assume this property as an OHA invariant and check for it dynamically as discussed in §3.7. When any branch conditional on an equality or non-equality comparison has any pointers in the comparison, we instrument the spatial safety invariant check for the pointer operands and then optimistically elide the implicit PPT operations. For any other logical comparison used in conditional branches, we always instrument all control-dependent value definitions.

4.2 Dynamic Pointer Provenance Tracking

After the static provenance analysis, we instrument the target program for dynamic Pointer Provenance Tracking (PPT) operations. This entails three types of pointer metadata operations, and OHA invariant checks.

Root-set Pointers: In an ideal world, Prov-GC would be able to identify all dynamic data in the root-set, statically, based on the state of the program at GC time, and finding the root set would require no instrumentation. However, as LLVM operates on an intermediate state, and doesn't

expose the mapping of this state to actual dynamic runtime state, Prov-GC requires some runtime instrumentation to identify the root set. To accomplish this, we instrument the main entry function to record the locations of all global values which may hold pointers. Every function, at entry, adds to this record the locations of all local pointer values and removes them before returning. This ensures that the GC can always locate statically identified safe pointer locations, and we only need to track the rest.

Pointer Metadata Creation: Any location that could not be statically identified as a non-pointer has two pieces of metadata associated with it- A pointer flag indicating that it has a safe pointer value, or its *dynamic points-to* set when it's unsafe. The metadata is split into two separate structures- a shadow memory with two bit taints (`00`:non-pointer, `01`:safe-pointer, `11`:unsafe-pointer) and the points-to set in a splay-tree [Sleator and Tarjan 1985] indexed by the pointer value location. The shadow memory taints are set when a safe pointer is assigned to a non-pointer or is loaded/stored from/to memory. When a safe pointer becomes unsafe, the original safe pointer's dynamic value is used to retrieve the object id from the GC Allocator Table (discussed later in §4.3). When the resultant unsafe pointer value is not in-bounds this object, the points-to set is created on-demand in the splay-tree and the shadow memory taint is set to `11`.

Pointer Metadata Tracking: The shadow memory taint creations, lookups and transfers are efficiently handled using LLVM DFSan [dfs] instrumentation. For explicit propagation of unsafe pointers, the points-to set metadata is computed as union of the sources. Since, most dynamic points-to sets are likely singleton, the set is represented as a tuple, and the first boolean item indicates if the second value is the singleton points-to value or a pointer to the points-to set. For implicit propagation, we use Iodine's recovery mechanism to create two paths, with and without the implicit tracking, based on the spatial safety invariant. In the instrumented path, all control dependent instruction values become unsafe and their points-to set accumulates that of the comparison operands.

OHA Invariant Checks: In addition to Iodine's [Banerjee et al. 2019] likely invariants used for information-flow analysis, our implicit provenance optimizations in §3.4 add additional spatial bounds checks for pointer comparison operands. These checks use the pointer's dynamic value to query the lower bound allocation base address and then check if it's within that allocation's bound address.

4.3 Garbage Collection

Prov-GC keeps an *allocation table* of active allocations with their base and bound addresses, indexed by their base address. This is exposed to the provenance tracking mechanism for computing the dynamic points-to and enforcing the spatial safety invariant checks.

Prov-GC uses the dynamic points-to and taint metadata maintained by PPT to compute the pointers-within set for a given object bounds efficiently. First, the splay tree representation of the Points-to set facilitates fast range queries, so that given an objects' bounds, locations within its bounds with non-empty points-to sets can be efficiently computed. This yields all the unsafe pointers within an object. Next, the one-to-one mapping of heap locations to their taint bits in shadow memory enables efficiently searching for and computing offsets of the safe pointers within the objects' bounds using bitwise arithmetic.

Garbage collection begins by pushing the root-set of pointers maintained by our root-set tracking into a set known as the GC root-set. Then it queries the allocation table to locate remaining pointers within the bounds of the global data segment and the current stack to include in the GC root-set. Then marking continues by transitively performing the range-queries into the bounds of the objects in their points-to set. The range-based query techniques quickly locate all pointer values within

an object's bounds, much faster than value-based scanning for large objects. This significantly improves the performance of Prov-GC.

Finally, when Prov-GC collects objects, it needs to additionally remove the metadata associated with them. The PPT module provides callbacks to Prov-GC to remove the Points-to metadata through a range-deletion operation, and also resets the shadow memory taints for the objects' bounds.

4.4 Source Transformations for GC

Running target programs with GC require some source-level changes to communicate between the collector and the client program. To convey applications' allocation requests, we replace all allocation calls e.g. `malloc()` with corresponding `GC_malloc()`'s, and remove all `free()`'s. Some applications like `redis`, need to be notified by the GC for special handling of deallocated objects, for which we use BDW-GC interface to register the application specific finalization code.

5 EVALUATION

Our evaluation shows the following:

- Dynamic Pointer Provenance Tracking incurs reasonably low overheads, even for memory-intensive benchmarks.
- Prov-GC reduces scanning overheads so that individual GC invocations run faster compared to BDW-GC.
- Prov-GC is the only sound garbage collector, and yet improves memory reclamation rates over value-based GC and yields benefits similar to other unsound GC solutions [Rafkind et al. 2009].

5.1 Experimental Setup

We evaluate Prov-GC over several real-world applications and benchmark programs including:

- SPECint 2006 C benchmarks: `perlbench`, `bzip2`, `gcc`, `mcf`, `gobmk`, `hmmer`, `sjeng`, `libquantum`, `h264ref` [Henning 2006].
- `nginx` web server [ngi] serving static webpages.
- `redis` database server [red] performing key-value store, list operations, and geographic search [til].
- `postfix` mail server [pos] running performance test generators.
- `vim` running pattern search and text processing [vim].

We run `nginx` serving pydoc3 documentation and loading several webpages; `redis` benchmarking application and performing geo-search [til]; the `postfix` stress tests; `vim` challenge solutions from [vim]; SPEC benchmarks with their reference inputs. Our static analysis is optimized using optimistic hybrid analysis [Devecsery et al. 2018]. So first, we profile a program over a set of profiling executions to gather likely invariants which we then use in a predicated static analysis to construct our final optimized dynamic provenance tracking analysis. Our profiling methodology is adopted from that used in Iodine [Banerjee et al. 2019], an optimistically optimized taint tracking tool, where we generate a diverse set of inputs per program by sweeping the programs' parameter space. We gather invariants by running on a *profile set* consisting 400 executions and also on regression test suites for `nginx`, `redis` and `vim`. All provenance tracking and GC experiments run on a separate *test set* of 100 executions. The one-time costs of profiling are far outweighed by the benefits of optimistic analysis.

Table 1. Benchmark configurations

Program	base	bdw-gc	
	Peak Memory	Heap Limit	# Collections
perlbench	580 MB	1024 MB	2
bzip2	856 MB	1024 MB	3
gcc†	940 MB	1024 MB	×
mcf	832 MB	1024 MB	3
gobmk	32 MB	32 MB	2
hmmer	60 MB	48 MB	1
sjeng	180 MB	256 MB	2
libquantum	108 MB	128 MB	2
h264ref	68 MB	48 MB	2
nginx*	26 MB	16 MB	2
redis*	316 MB	512 MB	3
postfix	588 MB	1024 MB	4
vim	244 MB	512 MB	3

†we're unable to run gcc with bdw-gc, *nginx and redis employ their own custom GC allocators

For each target program, we create 3 versions: (1) **base** without GC, (2) **bdw-gc** with value-based BDW-GC and (3) **prov-gc** with our sound Prov-GC. The base versions use `glibc 2.26` allocator, except for `nginx` and `redis` which use their own custom allocator wrappers that use `jemalloc 5.1.0`. All programs are compiled with `clang 7.0` at the `-O3` optimization level. For the `bdw-gc` versions, we replace calls to allocator functions (e.g. `malloc()`) with corresponding GC allocator functions (e.g. `GC_malloc()`) and omit all deallocations (i.e. `free()`) and use BDW-GC version `7.4.16` built without thread support, and with the parallel and incremental collection being disabled (`GC_MARKERS=1 GC_DISABLE_INCREMENTAL`). We use different heap limits as listed in Table 1 to trigger sufficient number of GC invocations as the programs exhibit widely varying memory footprints and programs like `nginx` are designed for low memory overhead. We were unable to run `gcc` using BDW-GC, so we exclude this program from the GC results §5.3 onward. Finally, the `prov-gc` versions compile with our static analysis that instruments them with the provenance tracking mechanism, and run with the Prov-GC allocator using the same configurations as above.

All experiments are run on a single core of an Intel Xeon E5-2620 processor with 64GB RAM running Linux 4.18.

5.2 Provenance Tracking Overheads

To understand how static analysis can significantly reduce the overhead of provenance tracking, we run Prov-GC configured only to track provenance (i.e. collection disabled), and then selectively enable optimizations within Prov-GC. Our results can be found in Figure 5: each benchmark shows 4 different overheads normalized to `base`– 'Cons' uses the sound static analyses described in §3.3 and §3.4 to optimize provenance tracking; 'Opt' further optimizes using optimistic hybrid analysis as described in §3.7; the two '+C' versions then use the specific optimization in §3.6 leveraging the C Standard. We find that the overhead of provenance tracking, including implicit flow tracking, for our benchmarks is actually quite reasonable, with an average overhead of 16% (11% excluding `gcc`). This result is actually quite surprising, as this number includes the cost of implicit flow tracking, which is known for dramatically increasing taint tracking overhead due to over-tainting. However, with our combination of static analyses, and optimistic hybrid analyses, we are able to dramatically reduce this result to only 16%. Note that this solution requires strict adherence to the C standard, which is stronger than spatial memory safety. While spatial memory safety only checks that pointers are in-bounds when dereferenced, the standard requires that pointers be in-bounds *always* for well-defined behavior. Therefore, the design point that does not assume the C Standard

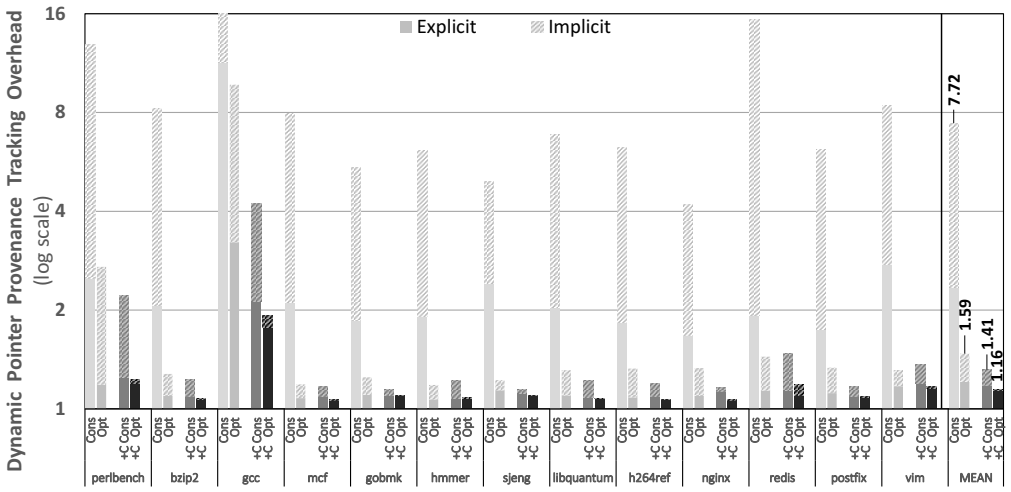

Fig. 5. prov-gc dynamic pointer provenance tracking overheads

Cons: optimizes dynamic provenance tracking using sound static analyses, Opt: uses optimistic hybrid analysis.
+C: optimizes for C Standard-compliant programs as discussed in §3.6.
The solid portions of bars represent the overheads of tracking provenance via explicit flows, and the striped portions represent that for implicit flows. Execution times are normalized to base, i.e. without GC.

is also quite useful in supporting sound GC for legacy non-portable C programs that do not follow this strict standard. Programs that violate the standard [Chisnall et al. 2015; Memarian et al. 2016] can still employ sound GC, although incurring a higher overhead of ~ 60% (37% excluding gcc). Note that this cost is still comparable to that of Temporal Memory Safety checking solutions (60% overhead [Zhang et al. 2019]).

gcc is a very large program for which whole program context-sensitive pointer analysis does not scale, even when predicated using optimistic hybrid analysis. The context-insensitive pointer analysis for this program results in much less precise may-alias relations. Consequently, non-pointers can be imprecisely classified as may-be pointers, and safe pointers to be unsafe. This induces much weaker static optimizations resulting in severe dynamic overheads. The average overhead of provenance tracking excluding gcc is 11%.

We further study the various sources of our provenance tracking overheads in detail. For explicit provenance tracking, ~ 21% of its overhead attributes to the provenance metadata creation– i.e. when a safe pointer becomes unsafe, we compute the provenance metadata from the value of the source safe pointer. This substantiates that very few pointer values become tainted as unsafe and the remainder of the overhead is in tracking their provenance propagation. On the contrary, for implicit provenance tracking, the overhead is entirely in validating the invariant of spatial bounds checking for pointer comparisons, as we discussed in §3.4, and none of our tested programs ever violate these checks. The framework overheads of checking the other optimistic invariant assumptions are negligibly low.

Summary. : Optimistic hybrid analysis combined with our optimizations in §3.3 significantly elides tracking operations for most non-pointers and safe pointers, and tracks only few unsafe pointers. Pointer provenances do not propagate implicitly via common pointer comparisons as we reasoned in §3.4, and checking for that involves a subset of spatial memory safety check overheads. The cost of soundness for GC in our Pointer Provenance Tracking is ~ 16%. This is significantly lower than the cost of providing Temporal Memory Safety (~ 60% [Zhang et al. 2019]).

Note that, to the best of our knowledge, these benchmarks do not exercise unsafe pointer propagations and BDW-GC also works correctly for them. For the programs that exercise unsafe pointer manipulations, BDW-GC might break, but Prov-GC still works correctly. However, programs with pathological pointer manipulations may incur higher overheads with Prov-GC. So, Prov-GC provides soundness in all cases without hurting performance significantly in the common case.

5.3 GC Overheads

Next, we show how the presence of dynamic pointer provenance information can be used to achieve an efficient GC solution. To study this effect, we compare the collection times of a single GC invocation with `prov-gc` against that of `bdw-gc`. Since our pointer provenance metadata is maintained separately outside the GC managed heap, the first GC invocation of a program happens at the same execution point under consistent configurations. But, since `bdw-gc` can reclaim unsoundly and retain imprecisely, the subsequent GC invocations can happen at different program states. So for equivalent comparison, we only measure collection statistics upon the first GC invocation and then terminate the program.

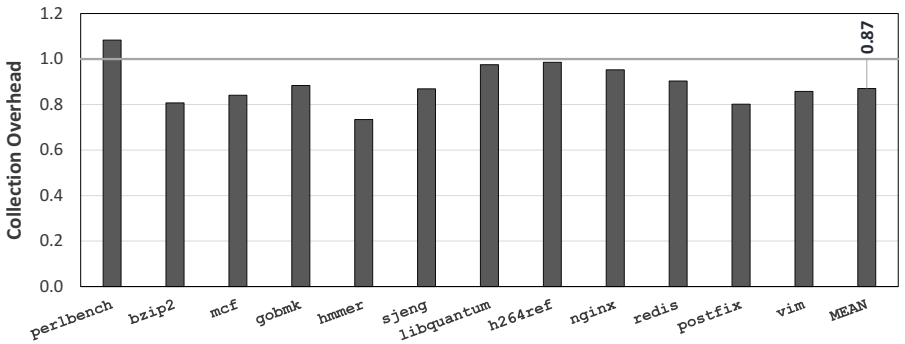

Fig. 6. `prov-gc` reduces overhead of a single GC invocation; collection time is normalized to that of `bdw-gc`

Figure 6 plots the overhead of a GC invocation with `prov-gc` normalized to that with `bdw-gc`. `prov-gc` generally improves the GC collection times compared to `bdw-gc` and completes collecting $\sim 13\%$ faster. While `bdw-gc` performs expensive value-based scanning over large allocations to locate potential pointers for marking, `prov-gc` can locate values with pointer provenance using the fast range queries over its metadata. This benefit can compound while marking large allocations with sparsely located pointers.

5.4 GC Effectiveness

We study the effectiveness of provenance-based GC in terms of its memory retention rate, i.e. the fraction of heap size after and before a GC invocation, once again in our previous single GC invocation setup. Figure 7 plots the mean memory retention for each program. `prov-gc` is strictly more precise than `bdw-gc` and reclaims as much as 21% more memory in the case of `vim`, and $\sim 6\%$ more on average. This benefit varies with the programs' memory usage patterns, and is low for programs with a stable working set like `mcf`, `hmmer` and `sjeng`. Prior works on Precise GC [Rafkind et al. 2009] for C report better reclamation (up to $\sim 70\%$) benefits on a different set of applications, although they do not guarantee that the reclaimed objects will not be used in the future.

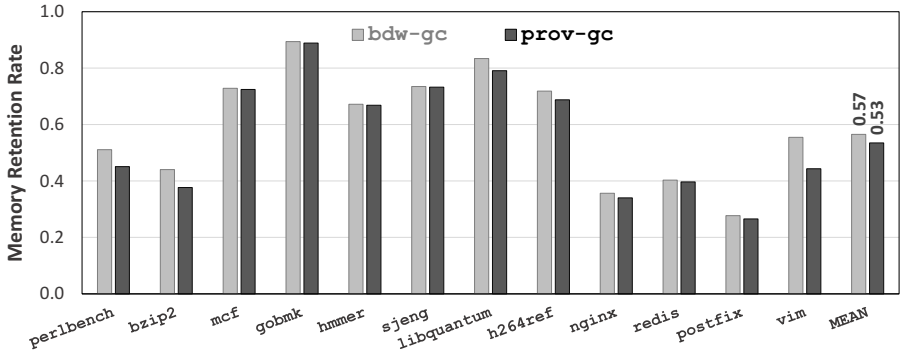

Fig. 7. prov-gc reclaims more memory per GC invocation

5.5 GC Heap Size Sensitivity

Finally, we look at the end-to-end performance of prov-gc and its space-time trade-off with varying heap sizes. Figure 8 shows the execution times of prov-gc and bdw-gc normalized to that of base for four benchmark programs with varying heap size limits on the x-axes; the labeled numbers on the plots indicate the number of GC invocations per benchmark configuration. Overall, prov-gc runs slower than bdw-gc and the difference between their execution time plots is attributed to the dynamic Pointer Provenance Tracking overheads. We observe that more frequent GC invocations at lower heap size limits lead to higher execution time overheads. This behavior is consistent for both bdw-gc and prov-gc although interestingly prov-gc's improved reclamation rate results in fewer GC invocations for vim at lower heap limits.

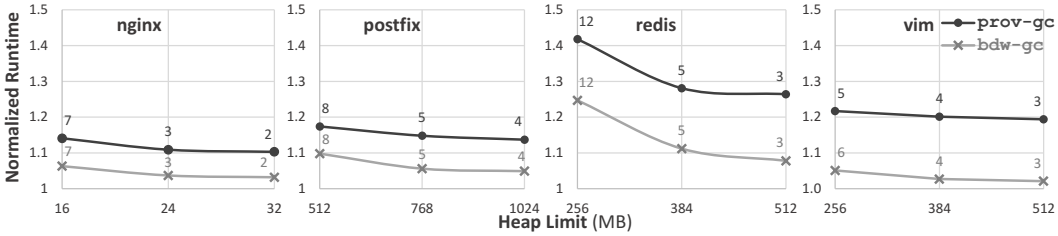

Fig. 8. prov-gc performance with varying heap limits

5.6 Memory Overheads

To evaluate the additional memory overheads of maintaining the pointer metadata, we measure the sizes of the two metadata structures- shadow memory, and splay tree, at collection time. Programs running with Prov-GC have 30.8% more memory footprint on average, with perlbench seeing the highest overhead of 35.1%. Of this memory overhead, ~ 27.1% is due to the shadow memory structure, and ~ 3.6% is occupied by the splay tree structure.

Tracking pointer provenance metadata incurs some additional memory overheads, and hence may not be suitable for applications with large memory footprints in memory constrained environments. However, the advantages of sound garbage collection are much prominent in contrast.

6 LIMITATIONS

Possible leaks: All GCs are imprecise and can lead to memory leaks. Prov-GC too may suffer from memory leaks in pathological cases where it finds benign data-flows from pointer values, e.g. when certain bits of `malloc` return addresses are checked for bookkeeping purposes. In practice we do not observe such behavior.

Thread safety: Prov-GC's implementation currently supports single-threaded programs. Future work can address this limitation by combining prior work on data-flow analysis for concurrent programs [Chugh et al. 2008] with our optimistic hybrid analysis techniques to construct a static pointer provenance analysis for concurrent programs. The C11 standard, by requiring data-race freedom, allows extending the sequential reasoning of many program analyses to concurrent programs [Effinger-Dean et al. 2011]. Note that the provenance metadata accesses do not introduce any new races, and the per-word taint metadata is already covered by the program's existing synchronization for shared objects.

7 RELATED WORK

In §2, we discussed the limitations of well-known value-based GC for C w.r.t soundness, precision, and GC performance. Below we relate relevant GC work that address each of these problems to our work. We also discuss work that relates to the techniques (provenance) we use and the added benefits of GC (temporal safety).

GC Soundness: Prior work has developed compiler checks to reject C/C++ programs that may violate soundness of value-based GC. Precise GC solutions check that programs do not store pointer values into integral types [Henderson 2002; Rafkind et al. 2009] during compilation. Conservative GC [Boehm 2004] checks its assumptions (e.g., that integers are not converted to heap pointers), and preserve original pointer values around compiler optimizations of pointer arithmetic [Boehm 1996; Boehm and Chase 1992]. However, these solutions can reject legal C/C++ programs, because they essentially make pointer manipulations and casting illegal.

GC Efficiency: Traditional mark-sweep collection [Boehm 2004] has been optimized using *parallel* marking algorithms [Boehm et al. 1991; Endo et al. 1997], by collecting *incrementally* [Baker and Hewitt 1977; Deutsch and Bobrow 1976], by treating *generations* of objects separately [Ungar 1984; Yip 1991], or by organizing the heap into regions and performing mark-region GC [Blackburn and McKinley 2008; Endo et al. 1997]. The fragmentation problem when dealing with ambiguous pointers in uncooperative environments like C has been addressed by *mostly-copying* collectors that move heap objects with no direct references from the root set [Bartlett 1988, 1989; Hosking 2006; Smith and Morrisett 1998; Yip 1991]. These optimizations are orthogonal to our goal to realize sound GC for C, but they can be integrated into our GC.

GC Precision: Precise GC techniques disambiguate pointers from non-pointers to some degree, relying on programmer annotations to register live pointers in a shadow stack to be managed by GC libraries [Edelson 1990; Edelson and Pohl 1991; Schreiner 1996], or with cooperation from the compilers [Jones et al. 1999]. When compiling high-level languages to C [Henderson et al. 1995; Peyton Jones et al. 1993; Tarditi et al. 1992], a virtual machine with its own stack and registers convey necessary type information to the GC, but this complicates code generation and makes systems fragile and non-portable. As inefficiencies of conservative collection arise mostly due to their conservative treatment of the root pointers [Hirzel et al. 2002], type-accurate GC [Henderson 2002] accurately locate pointers in a shadow stack through extensive source transformations. Later systems [Baker et al. 2007; Jung et al. 2006; Rafkind et al. 2009] improve upon this by optimizing metadata storage, and using static liveness analysis assuming the programs obey several constraints.

Such techniques are primarily motivated to solve the leakage and fragmentation issues by enabling copying collection, although they incur significant additional framework overheads [Henderson 2002; Rafkind et al. 2009] and their reduced retention and compaction benefits are marginal [Shahriyar et al. 2014]. But importantly, these systems are more unsound than conservative GC in assuming that pointer values are only stored into pointer types.

Reference Counting: In contrast to above reachability-based tracing collectors, Reference Counting keeps count of incoming references to each object [Collins 1960] and reclaims an object when its reference count falls to zero. This is natively supported in many languages like- Objective-C, Perl, PHP, and Swift, and also in C++ via 'smart pointers'. This requires the compiler to identify all pointer updates and account reference counts which can become expensive; so it is deferred and performed periodically and incrementally [Deutsch and Bobrow 1976]. Reference counting is inefficient compared to precise GC [Shahriyar et al. 2012, 2013], and moreover cannot collect cyclic garbage requiring separate cycle collectors [Bacon and Rajan 2001; Lins 1992] or forbidding cycles altogether [Apple 2013]. Importantly, they inherently rely on type safety and cannot handle pointer information flowing into non-pointers. Our dynamic pointer provenance could be used to maintain reference counts to objects, and thereby realize a sound reference-count based GC for C.

Taint and Provenance Analysis. : There is a significant body of work on static and dynamic information-flow analysis by tracking taints. They have been largely used to ensure that private data do not leak through untrusted channels [Newsome and Song 2005]. Static analysis [Sabelfeld and Myers 2003], including optimistic hybrid analysis [Banerjee et al. 2019; Devecsery et al. 2018], has been used to reduce the overhead of dynamic-taint tracking. While we leverage this body of work, our static analysis elides runtime monitors by taking advantage of special properties that is true only in dynamic pointer provenance. Also, unlike classical taint solutions, we show how implicit-flow tracking is necessary and feasible to track in our context.

Recent work has introduced static pointer provenance for C [Memarian et al. 2019] in order to improve static alias analysis. This static analysis was used by C compilers to improve compiler optimizations. In contrast, we discuss dynamic pointer provenance, and optimistic hybrid analysis to optimize it. Recent work used dynamic pointer provenance for implementing capability checks [Davis et al. 2019] to improve security. In contrast, we use dynamic pointer provenance to construct sound GC for weakly typed languages, and optimize that using optimistic hybrid analysis.

Memory Safety. : *Temporal memory safety* ensures that programs access only allocated memory, and *Spatial memory safety* ensure that all accesses are within allocated object bounds. Spatial MS is ensured by dynamically checking that intermediate pointer arithmetic do not cross valid object boundaries [Jones and Kelly 1997], and further tracking their intended objects for out-of-bounds pointers [Ruwase and Lam 2004]. This approach has been improved by allocating memory in pools and storing object bounds more efficiently [Dhurjati et al. 2003], by restricting memory allocation sizes and layout to efficiently compute bounds checks [Akritidis et al. 2009], eliminating redundant checks through static analysis [Bodík et al. 2000; Simpson and Barua 2013], and hybrid solutions combining static analysis with hardware support [Nagarakatte et al. 2009]. Temporal MS requires tracking liveness of objects and checking for erroneous uses of uninitialized objects and dangling pointers (use-after-free, double-free), which has also been heavily optimized using static analysis [Nagarakatte et al. 2010; Simpson and Barua 2013; Zhang et al. 2019].

Another approach to MS, adopted by Cyclone [Jim et al. 2002], CCured [Necula et al. 2002], SafeCode [Dhurjati et al. 2003], Checked C [Elliott et al. 2018], and Managed C++ [Microsoft 2004], is to enforce constraints through a strong type system and then perform sound analysis to check for memory errors, but the language becomes much restricted than C, making porting programs hard.

A contrasting approach is to combine heuristics, programmer annotations, and unsound analyses in designing tools [Austin et al. 1994; Ding and Zhong 2002; Dor et al. 1998; Evans 1996; Heine and Lam 2003; Sparud 1993] that detect most memory usage errors in practice.

Our sound GC guarantees temporal MS for legacy C/C++ code by automatically collecting safe garbage that cannot be accessed later, and we show this can be done more efficiently.

Dynamic Type Inference. : Types can be inferred from binaries during execution [Caballero and Lin 2016; Lee et al. 2011], optionally aided with static analysis [Elwazeer et al. 2013], for many applications including- decompilation, binary rewriting, vulnerability detection, memory forensics, etc. The typed binary can be executed with dynamic type-safety checks [Burrows et al. 2003]. Our dynamic tracking infers more than pointer types, as it also tracks the pointer provenance when type-safety is violated.

8 CONCLUSION

Traditionally, Garbage Collectors have relied only on values to identify pointers in uncooperative environments like C. C being weakly typed, this is unsound for several legal pointer manipulations. We show that tracking pointer provenances using Dynamic Information Flow Tracking can soundly identify all pointer information, even those hidden by their values, and a Provenance-based GC will therefore safely collect only objects which *cannot* be accessed later.

To realize a practical *Pointer Provenance Tracking* solution, we leverage Optimistic Hybrid Analysis, and identify properties that allow us to elide tracking for most common pointer operations. Our tool Prov-GC tracks pointer information propagations with only $\sim 16\%$ overhead, even via implicit control-flows, and also improves the overhead and effectiveness of collection.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their thoughtful comments. This work was supported by the National Science Foundation under grant SHF-1703931. The views and conclusions contained in this paper are solely those of the authors.

REFERENCES

. DFSan. Clang DataFlowSanitizer. http://clang.llvm.org/docs/DataFlowSanitizer.html.
. The NGINX web server. https://www.nginx.com.
. Optimistc Hybrid Analysis – LLVM tools. https://github.com/ddevec/OhaLLVM.
. The Postfix mail server. http://www.postfix.org.
. The Redis database server. https://redis.io.
. The Tile38 geolocation information systems. http://tile38.com.
. VimGolf. https://vimgolf.com.
2004. A Memory-Efficient Doubly Linked List. https://www.linuxjournal.com/article/6828.
Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, June 1990*, Jean-Loup Baer, Larry Snyder, and James R. Goodman (Eds.). ACM, 2–14. https://doi.org/10.1145/325164.325100
Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
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 *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, Fabian Monrose (Ed.). USENIX Association, 51–66. http://www.usenix.org/events/sec09/tech/full_papers/akritidis.pdf
Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Detecting Equality of Variables in Programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and P. Mager (Eds.). ACM Press, 1–11. https://doi.org/10.1145/73560.73561

Apple. 2013. Transitioning to ARC release notes. https://developer.apple.com/library/archive/releasenotes/ObjectiveC/RN-TransitioningToARC.

Thomas H. Austin and Cormac Flanagan. 2009. Efficient purely-dynamic information flow analysis. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, Stephen Chong and David A. Naumann (Eds.). ACM, 113–124. https://doi.org/10.1145/1554339.1554353

Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa (Eds.). ACM, 290–301. https://doi.org/10.1145/178243.178446

David F. Bacon, C. Richard Attanasio, Han Bok Lee, V. T. Rajan, and Stephen E. Smith. 2001. Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, Michael Burke and Mary Lou Soffa (Eds.). ACM, 92–103. https://doi.org/10.1145/378795.378819

David F. Bacon and V. T. Rajan. 2001. Concurrent Cycle Collection in Reference Counted Systems. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2072)*, Jørgen Lindskov Knudsen (Ed.). Springer, 207–235. https://doi.org/10.1007/3-540-45337-7_12

Henry G. Baker and Carl Hewitt. 1977. The incremental garbage collection of processes. *SIGART Newsl.* 64 (1977), 55–59. https://doi.org/10.1145/872736.806932

Jason Baker, Antonio Cunei, Filip Pizlo, and Jan Vitek. 2007. Accurate Garbage Collection in Uncooperative Environments with Lazy Pointer Stacks. In *Compiler Construction, 16th International Conference, CC 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 26-30, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4420)*, Shriram Krishnamurthi and Martin Odersky (Eds.). Springer, 64–79. https://doi.org/10.1007/978-3-540-71229-9_5

Subarno Banerjee, David Devecsery, Peter M. Chen, and Satish Narayanasamy. 2019. Iodine: Fast Dynamic Taint Tracking Using Rollback-free Optimistic Hybrid Analysis. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 490–504. https://doi.org/10.1109/SP.2019.00043

Joel F. Bartlett. 1988. Compacting Garbage Collection with Ambiguous Roots. 1, 6 (April 1988), 3–12. https://doi.org/10.1145/1317224.1317225

Joel F. Bartlett. 1989. *Mostly-copying garbage collection picks up generations and C++*. Technical Report TN. 12. Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA.

Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 158–168. https://doi.org/10.1145/1133981.1134000

Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 22–32. https://doi.org/10.1145/1375581.1375586

Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. 2000. ABCD: eliminating array bounds checks on demand. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, Monica S. Lam (Ed.). ACM, 321–333. https://doi.org/10.1145/349299.349342

Hans-Juergen Boehm. 1993. Space Efficient Conservative Garbage Collection. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 197–206. https://doi.org/10.1145/155090.155109

Hans-Juergen Boehm. 1996. Simple Garbage-Collector-Safety. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*, Charles N. Fischer (Ed.). ACM, 89–98. https://doi.org/10.1145/231379.231394

Hans-Juergen Boehm and David Chase. 1992. A Proposal for Garbage-Collector-Safe C Compilation. *The Journal of C Language Translation* 4, 2 (December 1992), 126–141. https://www.hboehm.info/gc/papers/boecha.ps.gz

Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. 1991. Mostly Parallel Garbage Collection. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, David S. Wise (Ed.). ACM, 157–164. https://doi.org/10.1145/113445.113459

Hans J. Boehm. 2004. Space Efficient Conservative Garbage Collection. *SIGPLAN Notices* 39, 4 (April 2004), 490–501. https://doi.org/10.1145/989393.989442

Michael D. Bond and Kathryn S. McKinley. 2006. Bell: bit-encoding online memory leak detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, John Paul Shen and Margaret Martonosi (Eds.). ACM, 61–72. https://doi.org/10.1145/1168857.1168866

Michael Burrows, Stephen N. Freund, and Janet L. Wiener. 2003. Run-Time Type Checking for Binary Programs. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2622)*, Görel Hedin (Ed.). Springer, 90–105. https://doi.org/10.1007/3-540-36579-6_7

Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, Mats Per Erik Heimdahl and Zhendong Su (Eds.). ACM, 133–143. https://doi.org/10.1145/2338965.2336769

Juan Caballero and Zhiqiang Lin. 2016. Type Inference on Executables. *ACM Comput. Surv.* 48, 4 (2016), 65:1–65:35. https://doi.org/10.1145/2896499

David Chisnall, Colin Rothwell, Robert N. M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. 2015. Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, 117–130. https://doi.org/10.1145/2694344.2694367

Ravi Chugh, Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2008. Dataflow analysis for concurrent programs using datarace detection. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 316–326. https://doi.org/10.1145/1375581.1375620

James A. Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, David S. Rosenblum and Sebastian G. Elbaum (Eds.). ACM, 196–206. https://doi.org/10.1145/1273463.1273490

James A. Clause and Alessandro Orso. 2010. LEAKPOINT: pinpointing the causes of memory leaks. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 515–524. https://doi.org/10.1145/1806799.1806874

Jacques Cohen. 1981. Garbage Collection of Linked Data Structures. *Comput. Surveys* 13, 3 (1981), 341–367. https://doi.org/10.1145/356850.356854

George E. Collins. 1960. A method for overlapping and erasure of lists. *Commun. ACM* 3, 12 (1960), 655–657. https://doi.org/10.1145/367487.367501

Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey D. Son, and Jonathan Woodruff. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 379–393. https://doi.org/10.1145/3297858.3304042

L. Peter Deutsch and Daniel G. Bobrow. 1976. An Efficient, Incremental, Automatic Garbage Collector. *Commun. ACM* 19, 9 (1976), 522–526. https://doi.org/10.1145/360336.360345

David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2018. Optimistic Hybrid Analysis: Accelerating Dynamic Analysis through Predicated Static Analysis. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar (Eds.). ACM, 348–362. https://doi.org/10.1145/3173162.3177153

Dinakar Dhurjati and Vikram S. Adve. 2006. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *2006 International Conference on Dependable Systems and Networks (DSN 2006), 25-28 June 2006, Philadelphia, Pennsylvania, USA, Proceedings*. IEEE Computer Society, 269–280. https://doi.org/10.1109/DSN.2006.31

Dinakar Dhurjati, Sumant Kowshik, Vikram S. Adve, and Chris Lattner. 2003. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03). San Diego, California, USA, June 11-13, 2003*, Frank Mueller and Ulrich Kremer (Eds.). ACM, 69–80. https://doi.org/10.1145/780732.780743

Chen Ding and Yutao Zhong. 2002. Compiler-directed run-time monitoring of program data access. In *Proceedings of The Workshop on Memory Systems Performance (MSP 2002), June 16, 2002 and The International Symposium on Memory Management (ISMM 2002), June 20-21, 2002, Berlin, Germany*, Hans-Juergen Boehm and David Detlefs (Eds.). ACM, 1–12. https://doi.org/10.1145/773039.773040

Nurit Dor, Michael Rodeh, and Shmuel Sagiv. 1998. Detecting Memory Errors via Static Pointer Analysis (Preliminary Experience). In *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering,*

PASTE '98, Montreal, Canada, June 16, 1998, Thomas Ball, Frank Tip, and A. Michael Berman (Eds.). ACM, 27–34. https://doi.org/10.1145/277631.277637

Daniel Ross Edelson. 1990. *Dynamic storage reclamation in C++.* Technical Report UCSC-CRL-90-19. University of California at Santa Cruz.

Daniel R. Edelson and Ira Pohl. 1991. A Copying Collector for C++. In *Proceedings of the C++ Conference. Washington, D.C., USA, April 1991*. USENIX Association, 85–102.

Laura Effinger-Dean, Hans-Juergen Boehm, Dhruva R. Chakrabarti, and Pramod G. Joisha. 2011. Extended sequential reasoning for data-race-free programs. In *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI '11, San Jose, CA, USA, June 5, 2011*, Jeffrey S. Vetter, Madanlal Musuvathi, and Xipeng Shen (Eds.). ACM, 22–29. https://doi.org/10.1145/1988915.1988922

Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development, SecDev 2018, Cambridge, MA, USA, September 30 - October 2, 2018*. IEEE Computer Society, 53–60. https://doi.org/10.1109/SecDev.2018.00015

Khaled Elwazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. Scalable variable and data type detection in a binary rewriter. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 51–60. https://doi.org/10.1145/2491956.2462165

Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. 1997. A Scalable Mark-Sweep Garbage Collector on Large-Scale Shared-Memory Machines. In *Proceedings of the ACM/IEEE Conference on Supercomputing, SC 1997, November 15-21, 1997, San Jose, CA, USA*. ACM, 48. https://doi.org/10.1145/509593.509641

David E. Evans. 1996. Static Detection of Dynamic Memory Errors. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*, Charles N. Fischer (Ed.). ACM, 44–53. https://doi.org/10.1145/231379.231389

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

Matthias Hauswirth and Trishul M. Chilimbi. 2004. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, Shubu Mukherjee and Kathryn S. McKinley (Eds.). ACM, 156–164. https://doi.org/10.1145/1024393.1024412

David L. Heine and Monica S. Lam. 2003. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, Ron Cytron and Rajiv Gupta (Eds.). ACM, 168–181. https://doi.org/10.1145/781131.781150

Fergus Henderson. 2002. Accurate garbage collection in an uncooperative environment. In *Proceedings of The Workshop on Memory Systems Performance (MSP 2002), June 16, 2002 and The International Symposium on Memory Management (ISMM 2002), June 20-21, 2002, Berlin, Germany*, Hans-Juergen Boehm and David Detlefs (Eds.). ACM, 256–263. https://doi.org/10.1145/773039.512449

Fergus Henderson, Zoltan Somogyi, and Thomas Conway. 1995. Compiling logic programs to C using GNU C as a portable assembler. In *Proceedings of The ILPSâĂŹ95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming Languages, Portland, Oregon.*

John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4 (2006), 1–17. https://doi.org/10.1145/1186736.1186737

Martin Hirzel, Amer Diwan, and Johannes Henkel. 2002. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Trans. Program. Lang. Syst.* 24, 6 (2002), 593–624. https://doi.org/10.1145/586088.586089

Antony L. Hosking. 2006. Portable, mostly-concurrent, mostly-copying garbage collection for multi-processors. In *Proceedings of the 5th International Symposium on Memory Management, ISMM 2006, Ottawa, Ontario, Canada, June 10-11, 2006*, Erez Petrank and J. Eliot B. Moss (Eds.). ACM, 40–51. https://doi.org/10.1145/1133956.1133963

ISO. 2018. *Programming languages — C (ISO/IEC 9899:201x).* Technical Report N2310. International Organization for Standardization, Geneva, Switzerland. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2310.pdf

Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, Carla Schlatter Ellis (Ed.). USENIX, 275–288. http://www.usenix.org/publications/library/proceedings/usenix02/jim.html

Richard W. M. Jones and Paul H. J. Kelly. 1997. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *Proceedings of the Third International Workshop on Automated Debugging, AADEBUG 1997, Linköping, Sweden, May 26-27, 1997 (Linköping Electronic Articles in Computer and Information Science, Vol. 2)*, Mariam Kamkar (Ed.). Linköping University Electronic Press, 13–26. http://www.ep.liu.se/ecp/article.asp?issue=001&article=002

Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. 1999. C–: A Portable Assembly Language that Supports Garbage Collection. In *Principles and Practice of Declarative Programming, International Conference PPDP'99, Paris, France, September 29 - October 1, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1702)*, Gopalan Nadathur (Ed.). Springer, 1–28. https://doi.org/10.1007/10704567_1

Maria Jump and Kathryn S. McKinley. 2007. Cork: dynamic memory leak detection for garbage-collected languages. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 31–38. https://doi.org/10.1145/1190216.1190224

Dong-Heon Jung, Sung-Hwan Bae, Jaemok Lee, Soo-Mook Moon, and Jong Kuk Park. 2006. Supporting precise garbage collection in Java Bytecode-to-C ahead-of-time compiler for embedded systems. In *Proceedings of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2006, Seoul, Korea, October 22-25, 2006*, Seongsoo Hong, Wayne H. Wolf, Krisztián Flautner, and Taewhan Kim (Eds.). ACM, 35–42. https://doi.org/10.1145/1176760.1176767

Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society. https://www.ndss-symposium.org/ndss2011/tie-principled-reverse-engineering-of-types-in-binary-programs

Rafael Dueire Lins. 1992. Cyclic Reference Counting with Lazy Mark-Scan. *Inform. Process. Lett.* 44, 4 (1992), 215–220. https://doi.org/10.1016/0020-0190(92)90088-D

Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C semantics and pointer provenance. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 67:1–67:32. https://doi.org/10.1145/3290380

Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the depths of C: elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 1–15. https://doi.org/10.1145/2908080.2908081

Microsoft. 2004. Managed Extensions for C++. https://docs.microsoft.com/en-us/cpp/build/reference/microsoft-extensions-to-c-and-cpp.

Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 245–258. https://doi.org/10.1145/1542476.1542504

Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, Jan Vitek and Doug Lea (Eds.). ACM, 31–40. https://doi.org/10.1145/1806651.1806657

George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: type-safe retrofitting of legacy code. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, John Launchbury and John C. Mitchell (Eds.). ACM, 128–139. https://doi.org/10.1145/503272.503286

James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*. The Internet Society. https://www.ndss-symposium.org/ndss2005/dynamic-taint-analysis-automatic-detection-analysis-and-signaturegeneration-exploits-commodity/

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, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 397–407. https://doi.org/10.1145/1542476.1542521

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. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 2 (2018), 28:1–28:30. https://doi.org/10.1145/3224423

Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. 1993. The Glasgow Haskell Compiler: a technical overview. In *Proceedings of UK Joint Framework for Information Technology (JFIT) Technical Conference, Keele*. 249–257.

Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. 2009. Precise garbage collection for C. In *Proceedings of the 8th International Symposium on Memory Management, ISMM 2009, Dublin, Ireland, June 19-20, 2009*, Hillel Kolodner and Guy L. Steele Jr. (Eds.). ACM, 39–48. https://doi.org/10.1145/1542431.1542438

Masoomeh Rudafshani and Paul A. S. Ward. 2017. LeakSpot: detection and diagnosis of memory leaks in JavaScript applications. *Software: Practice and Experience* 47, 1 (2017), 97–123. https://doi.org/10.1002/spe.2406

Olatunji Ruwase and Monica S. Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*. The Internet Society. https://www.ndss-symposium.org/ndss2004/practical-dynamic-buffer-overflow-detector/

Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

Wolfgang Schreiner. 1996. *RT++ - higher order threads for C++, tutorial and reference manual*. Technical Report 96-9. RISC-Linz. https://www3.risc.jku.at/software/rt++/

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 317–331. https://doi.org/10.1109/SP.2010.26

Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. 2012. Down for the count? Getting reference counting back in the ring. In *International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16, 2012*, Martin T. Vechev and Kathryn S. McKinley (Eds.). ACM, 73–84. https://doi.org/10.1145/2258996.2259008

Rifat Shahriyar, Stephen M. Blackburn, and Kathryn S. McKinley. 2014. Fast conservative garbage collection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 121–139. https://doi.org/10.1145/2660193.2660198

Rifat Shahriyar, Stephen M. Blackburn, Xi Yang, and Kathryn S. McKinley. 2013. Taking off the gloves with reference counting Immix. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 93–110. https://doi.org/10.1145/2509136.2509527

Matthew S. Simpson and Rajeev Barua. 2013. MemSafe: ensuring the spatial and temporal memory safety of C at runtime. *Software: Practice and Experience* 43, 1 (2013), 93–128. https://doi.org/10.1002/spe.2105

Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-Adjusting Binary Search Trees. *J. ACM* 32, 3 (1985), 652–686. https://doi.org/10.1145/3828.3835

Frederick Smith and J. Gregory Morrisett. 1998. Comparing Mostly-Copying and Mark-Sweep Conservative Collection. In *International Symposium on Memory Management, ISMM '98, Vancouver, British Columbia, Canada, 17-19 October, 1998, Conference Proceedings*, Simon L. Peyton Jones and Richard E. Jones (Eds.). ACM, 68–78. https://doi.org/10.1145/286860.286868

Jan Sparud. 1993. Fixing Some Space Leaks without a Garbage Collector. In *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*, John Williams (Ed.). ACM, 117–124. https://doi.org/10.1145/165180.165196

David Tarditi, Peter Lee, and Anurag Acharya. 1992. No Assembly Required: Compiling Standard ML to C. *LOPLAS* 1, 2 (1992), 161–177. https://doi.org/10.1145/151333.151343

David M. Ungar. 1984. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, USA, April 23-25, 1984*, William E. Riddle and Peter B. Henderson (Eds.). ACM, 157–167. https://doi.org/10.1145/800020.808261

John Vilk and Emery D. Berger. 2018. BLeak: automatically debugging memory leaks in web applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 15–29. https://doi.org/10.1145/3192366.3192376

Guoqing (Harry) Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. 2011. LeakChaser: helping programmers narrow down causes of memory leaks. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 270–282. https://doi.org/10.1145/1993498.1993530

G. May Yip. 1991. *Incremental, generational mostly-copying garbage collection in uncooperative environment*. Technical Report 91/8. Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA. https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-91-8.pdf

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, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 631–644. https://doi.org/10.1145/3297858.3304017