Pointer Ownership Model

Abstract

Pointers are a dangerous feature provided by C/C++, and incorrect use of pointers is a common source of bugs and vulnerabilities. Most new languages lack pointers or severely restrict their capabilities. Nonetheless, many C/C++ programmers work with pointers safely, by maintaining an internal model of when memory accessed through pointers should be allocated and subsequently freed. This model is frequently not documented in the program. The Pointer Ownership Model (POM) can statically identify certain classes of errors involving dynamic memory in C/C++ programs. It works by requiring the developer to identify responsible pointers, whose objects must be explicitly freed before the pointers themselves may be destroyed. POM can be statically analyzed to ensure that the design is consistent and secure, and that the code correctly implements the design. Consequently, POM can be used to identify, and eliminate many dynamic memory errors from C programs.

1. Introduction

Pointers are a powerful, but dangerous feature provided by the C and C++ programming languages, and incorrect use of pointers is a common source of bugs and security vulnerabilities. Most new languages lack pointers or severely restrict their capabilities, providing memory safety to eliminate these problems. Nonetheless, many C & C++ programs work with pointers safely. C pointers are still considered a powerful and useful feature of the language.

Programmers who safely work with pointers maintain an internal model of when memory accessed through those pointers should be allocated and subsequently freed. Commonly-applied models include garbage collection, Resource Acquisition Is Initialization (RAII), and smart pointers. The chosen model is frequently not documented in the program, because the language lacks any mechanism for encoding this information.

The pointer ownership model (POM) can statically identify certain classes of errors involving dynamic memory in C programs. The model works by requiring the developer to identify *responsible* pointers; these are the pointers whose objects must be explicitly freed before the pointers themselves may be destroyed. POM can be statically analyzed to ensure that the design is consistent and secure, and that the code correctly implements the design. Consequently, POM can be used to identify, and eliminate many dynamic memory errors from C programs.

The model is not perfect. It identifies several types of errors, including memory leaks, double-frees, and null dereferences. Out-of-bounds reads or writes to memory fall outside the current scope of the POM; hence it cannot detect buffer overflows. It can identify some, but not all use-after-free errors. Finally, the model provides an escape clause for the developer: any pointer may be marked as Out-Of-Scope (OOS), which means that no checking is performed on it. This is useful for pointers to dynamic memory that are handled by other ownership models, such as garbage collection.

2. Impact

This project aims to improve software security at the coding level. This would not only improve software security in general, but also improves productivity. It has been shown that eliminating bugs during the coding phase is much cheaper than eliminating them during the testing phase, or, even worse, after the bugs appear in production code. As POM could be considered an extension to the C language, POM would enhance the expressiveness of the C language to address coding challenges. Also it would achieve effective evaluation for a degree of memory safety.

The POM will employ several techniques that have been used by SEI research projects in the past. POM will require modification of a compiler (Clang/LLVM in particular). The *Safe/Secure C/C++* system [1] and *AIR Integers*[2], jointly developed by Plum Hall and SCI both rely primarily on static analysis, with dynamic analysis as a backup. Likewise, the C++ Thread Role Analysis proposal and the Compiler Enhanced Buffer Overflow Elimination proposal are both ongoing research projects that again rely primarily on static analysis, with dynamic analysis as a backup. (In fact, the Compiler Enhanced Buffer Overflow Elimination proposal implements the same safety

guarantees as *Safe/Secure C/C++)*. All of these projects also involve enhancements and modifications to the LLVM compiler framework.

Ultimately, we believe that adding a small judicious amount of annotations to a C program enables us to statically determine if the program is safe with regard to memory management. This means that the program need not be built or run. Such a strong guarantee will improve the maintainability of the program and consequently speed up software development.

3. Related Work

POM is related, and partially inspired, by several earlier research projects. Many software tools, both proprietary and open-source, employ pure static analysis to find memory-related errors, including both Fortify SCA and Coverity Prevent. These tools suffer from false positives; because they analyze code with no extra annotations or semantic information, they must make heuristic guesses about code correctness, and sometimes they guess wrongly.

Many other tools, such as Valgrind, provide dynamic analysis. They do not suffer from false positives, but they do not always find problems. To analyze a program, they require the developer to actually run the program, and if the program does not misbehave while being analyzed, dynamic analysis tools may not realize that it might misbehave when given a different set of inputs. Dynamic analysis also impedes performance, and so dynamic analysis tools are useless on programs that are being used in production; dynamic analyzers are typically only used by QA testers.

There are several attempts to improve memory management using models. The best known model for pointer management is reference counting, which is implemented in the C++11 standard library as the shared_pointer<> template. Garbage collection is an alternate model, which is used by the Java language, and there are several C implementations; however garbage collection is not part of standard C or C++.

Finally, the standard C++ auto_ptr<> template provides a similar model to our pointer ownership model; but it suffers from technical difficulties. The saga of the auto_ptr serves as an important cautionary tale when considering pointer ownership. In C++, the auto_ptr encoded a pointer that automatically freed its object upon its own destruction. It also supported assignment, with the stipulation that assigning one auto_ptr to another caused the first auto_ptr to be set to NULL…an unusual form of assignment that prevents 'perfect' duplication of owning pointers. This made auto_ptrs problematic particularly when used in containers, because containers assumed that their objects were fully copy-able, and auto_ptrs do not support this contract. The common implementation of the quick-sort algorithm would often fail with vectors containing auto_ptrs, because it typically involves assigning one single auto_ptr as a *pivot* and then partitioning the vector's remaining elements. Today the auto_ptr template has been deprecated in favor of the new **unique_ptr<>** template, which specifically forbids assignment or copying.

How can the Pointer Ownership Model avoid the mistakes that beset auto_ptr? We believe that POM can avoid these problems for two reasons. First, unlike auto_ptrs, POM is not required to strictly conform to the C/C++ type system. Assignment can be performed from one responsible pointer to another, or from a responsible pointer to an irresponsible pointer; based on the discretion of the developer. In contrast, auto_ptrs were constrained by the type system in certain cases. For example, assigning a variable to a vector item element requires that the variable also be an auto_ptr, causing the vector's corresponding auto_ptr to be set to null. Second, the POM adds semantic information to working code, but does not modify the code itself. As previously noted, assigning a variable to a vector of auto_ptrs causes the vector to be modified (because the corresponding vector item is nulled out). Under POM, the vector is not changed, although its item might be considered a zombie pointer. A quick-sort algorithm that operates on a vector of responsible pointers could use an irresponsible pointer as a pivot, thus requiring no changes to the algorithm code while still allowing pointer responsibility to be correctly monitored.

Pointer models have been used to address other problems, too. The *CCured* system[i] used a model of pointers, partially specified by the developer, to identify insecure pointer arithmetic, and catch buffer overflows. Likewise, the *Safe/Secure C/C++* system[ii][iii] uses both static and dynamic analysis to guarantee the safety of pointer arithmetic and array I/O. This enables the system to identify safe pointer usage at compile time, and add run-time checks when safety cannot be verified statically.

The POM is similar in that it performs safety checks at compile time, and could add run-time checks when necessary. Unlike traditional static analysis, POM imposes on the developer a requirement of providing some additional semantic information about the pointers used by the program. This extra information renders POM sound; any error it finds will be true positives. POM will also be comprehensive, finding all of the problems that are exposed by the model.

To be more precise, POM requires developers to specify a model for how memory is managed. After the

model is provided, POM can determine if the model is incomplete or inconsistent, and it can also determine whether the program complies with the model.

4. Design

The Pointer Ownership Model seeks to provide sound static analysis by using additional semantic information. Some of this information can be inferred from the source code, while some information must be explicitly provided by the developer. In particular, the model categorizes pointers into *responsible* pointers and *irresponsible* pointers. A responsible pointer is a pointer that is responsible for freeing its pointed-to object, and irresponsible pointer is not responsible for freeing whatever object it points to. Every object on the heap must be referenced by exactly one responsible pointer. Consequently, responsible pointers form trees of heap objects, with the tree roots living outside the heap.

Irresponsible pointers can point anywhere, but cannot free anything. A pointer variable is always responsible or irresponsible; it cannot change responsibility throughout its lifetime.

Ownership of an object can be transferred between responsible pointers, but when doing so, one of the pointers must relinquish responsibility for the object, as only one pointer may be responsible for freeing the object. Any responsible pointer that relinquishes responsibility for an object becomes a 'zombie' pointer; a compliant program will never use the value of any zombie pointer. Ideally, a responsible pointer's lifetime ends shortly after it becomes a zombie pointer.

Responsible pointers do not always point to objects they must free. Any responsible pointer can be in one of four states: good, null, uninitialized, and zombie. A good pointer is one that must be freed and only a good pointer may be dereferenced. A null pointer has the null value. It may be freed, but need not be freed; the standard guarantees that explicitly freeing a null pointer does nothing. An uninitialized pointer is one that does not hold a valid value; often because it is uninitialized. Finally, a zombie pointer can be a formerly good pointer that relinquished responsibility of an object to another good pointer. Or it could be a pointer that was passed to **free()**, and therefore no longer points to valid memory. Zombie pointers must not be dereferenced.

The POM project consists of two tools: an advisor and a verifier, as shown in Figure 1. The advisor examines a source code file and builds a model of pointer ownership, possibly with the help of the developer. (A sufficiently advanced developer could bypass the advisor entirely and build the model from scratch, but the purpose of the advisor is to save the developer from this work.) The verifier takes a compilation unit and a model and indicates if the model is complete and the code complies with it. The verifier can output errors if the model is incomplete (for instance, it doesn't indicate any responsibility for some pointer), or the model is inconsistent (it may say that a certain pointer is both responsible and irresponsible), or if the program violates the model (by trying to free an irresponsible pointer, for example.)

To analyze a compilation unit, the verifier requires that the model indicate the responsibility status of all pointers referenced by the compilation unit. However, pointers in the program that are not referenced by that compilation unit need not be specified. The model can be built alongside the verification of individual compilation units. As such, POM does analysis on the level of the compilation unit, rather than whole program analysis.

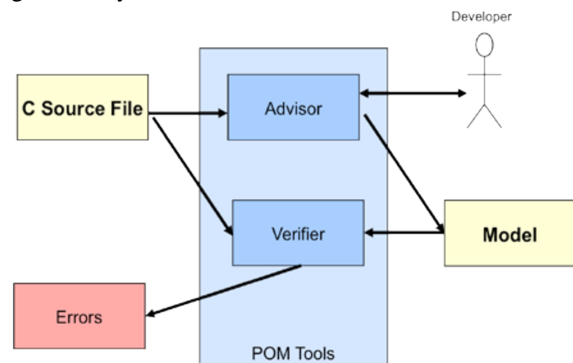

Figure 1. POM Implementation

4.1 Specification

All declared pointers can be in one of several subtypes, as listed in Table 1.

Table 1 Pointer Subtypes

Type	Definition	Examples
Unknown	Hasn't been assimilated into POM yet	
Out Of Scope (OOS)	POM doesn't apply to this pointer; it is handled some other way.	• Reference-counting • Garbage collection • Circular linked list • FILE pointers (could be considered distinct resources under a model similar to POM
Irresponsible	Not responsible for cleaning up memory it points to	Any pointer pointing into the stack or data segment
Responsible	Responsible for cleaning up memory it points to	Any pointer that gets the result of a call to malloc()

These pointer subtypes are subject to the following rules:

- A POM-compliant program shall not have any unknown pointers.
- An OOS pointer shall not be assigned the value of a responsible pointer.
- An irresponsible pointer should never be a producer argument or producer return value.

Example:

```
char* irp = INITIALIZE_IRRESPONSIBLE;
irp = malloc(5); // bad, malloc's return is a producer
```

- An irresponsible pointer should never be a consumer argument.

Example:

```
char* irp = INITIALIZE_IRRESPONSIBLE;
free(irp); // bad, free consumes its argument
```

- An irresponsible pointer should never by copied over to a responsible pointer.

Example:

```
char* rp; // responsible
char* irp = INITIALIZE_IRRESPONSIBLE;
rp = irp; // bad
memcpy( &rp, &irp, sizeof( rp)); // bad
```

- A responsible pointer should never get assigned a value from pointer arithmetic

Example:

```
char *rp1 = ...; // responsible
char *rp2; // responsible
rp1 = rp2 + 1; // bad
```

- A responsible pointer should never get assigned a value created by &

Example:

```
char *rp1 = ...; // responsible
char *rp2; // responsible
rp1 = &rp2; // bad
```

In other words a responsible pointer can be assigned only the following:

- a producer return value (or be supplied as a producer argument)

Example:

```
char *rp1; // responsible
char *rp2; // responsible
rp1 = malloc(5);  // ok
getline( &rp2, 80, stdin); // ok
```

- another responsible pointer

Example:

```
char *rp1 = ...; // responsible
char *rp2; // responsible
char *rp3; // responsible
rp1 = rp2; // ok
memcpy( &rp3, &rp1, sizeof( rp3)); // ok
```

- Null

Example:

```
char *rp; // responsible
rp = NULL; // ok
```

Responsible pointers can be in one or more of the following states. The state may not always be known, in which case the pointer can occupy multiple states, and be resolved later.

- Uninitialized (which includes any invalid value)

Example:
```
char *ptr; // uninitialized
char *ptr2 = ptr;  // also uninitialized
memcpy( &ptr2, &ptr, sizeof( ptr)); // still uninitialized
```

  - An uninitialized responsible pointer's value shall never be copied to an irresponsible pointer

Example:
```
char *rp; // responsible
char *rp2; // responsible
char *irp; // iresponsible
irp = rp; // bad
rp2 = rp; // ok
```

  - An uninitialized responsible pointer shall never be consumed

Example:
```
char *rp; // responsible
free( rp); // bad
```

  - An uninitialized responsible pointer shall never be dereferenced

Example:
```
char *rp; // responsible
*rp = '\0'; // bad
```

- Null
  - A responsible null pointer shall never be dereferenced

Example:
```
char *rp = NULL; // responsible
*rp = '\0'; // bad
```

- Good (that is, the pointer points to memory it must free, and no other responsible pointer points to this memory)

Example:
```
char *ptr = new char[5]; // good
char *ptr2 = malloc(5);  // good or null
if (ptr2 == NULL) abort();
// if we get here, ptr2 is good
```

  - A good pointer shall never be overwritten

Example:
```
char *rp = new char[5]; // responsible
rp = new char[7]; // bad
```

  - A responsible pointer shall never go out of scope while good

Example:
```
{
  char *rp = malloc(5); // responsible or null
}  // bad, memory might be leaked
```

```
struct foo_s {
  char* c; // responsible
} *s;
s = new struct foo_s;
s->c = new char[3];
// ...
delete s; // bad, s->c's memory lost
```

- Zombie (the pointer value has either been assigned to another responsible pointer, or freed)

Example:
```
free( ptr); // zombie
ptr = ptr2; // ptr2 now zombie, ptr now good
```

  - A zombie pointer's value shall never be read

Example:
```
char *rp = malloc( 5); // responsible, good or null
free( rp); // zombie or null
char *rp2 = rp; // bad
```

  - A zombie pointer shall never be consumed

Example:
```
char *rp = malloc(5); // responsible, good
free(rp); // zombie
free(rp); // bad (double free)
```

  - A zombie pointer shall never be dereferenced

Example:
```
char *rp = malloc(5); // responsible, good
free(rp); // zombie
rp[0] = '\0'; // bad
```

The subtypes of pointer variables never changes; once a responsible pointer, always a responsible pointer. However, the state of responsible pointers can change in the following ways:

- Assignment (this includes any read of one responsible pointer and write of the value into another responsible pointer)

Example:
```
char *rp1 = ...; // responsible
char *rp2;
rp2 = rp1; // rp1 gets rp1's state.
//  If rp1 was good, it is now a zombie
memcpy( &rp1, &rp2, sizeof( rp1));
// rp1 gets rp2's state.
// if rp2 was good, it is now a zombie
```

- Producer Return Value (this makes pointer good (or null, under some circumstances))

Example:
```
char *rp1; // responsible
rp1 = malloc(5); // rp1 is now good or null
```

```
char *rp2; // responsible
rp2 = new char[5]; // rp2 is now good
```

- Producer Argument (this makes pointer good (sometimes only if pointer was null)

Example:
```
char *rp = NULL; // responsible, null
getline( &rp, 80, stdin); // rp now good
```

- Consumer Argument (this makes pointer a zombie if and only if it was good)

Example:
```
char *rp = ...; // responsible, good
free(r); // rp now zombie
```

- Null check (A pointer that may be in several states can be disambiguated by a null check)

Example:
```
char *rp = malloc(5); // responsible, good or null
if (rp == NULL) {
  // rp is now null
  abort(); // or any noreturn function
} else {
  // rp is now good
}
// rp is now good
```

Like pointers, functions can be annotated when they take pointer arguments or when they return a pointer. Functions can be annotated with the following:

- Noreturn functions (These are functions that never return to their caller. They always exit via abort(), exit(), _Exit(), longjmp(), goto, or by invoking another noreturn function. The C11 _Noreturn keyword denotes noreturn functions.

Example:
```
void panic(const char *s) { // noreturn function
  printf("Error: %s\n", s);
  abort();
}
```

- Producer return functions (These are functions that return a pointer to a responsible object.)

Example:
```
malloc // null or good value
strdup // null or good value
operator new  // good value (never null)
```

  - Any producer return function must have its return value assigned to a responsible pointer

Example:
```
malloc(5); // bad, return value discarded
```

- Consumer arguments (These are arguments to certain functions. They either free the pointer's data, or give it to some other responsible pointer.)

Example:
```
free // arg 0: null or good -> zombie
```

  - A consumer argument must always be provided by a responsible pointer

Example:
```
char *rptr = ...; // responsible, good
free(ptr[3]) // bad
```

- Producer arguments (These are arguments to certain functions. They are of type 'pointer to pointer to x', and the functions generally allocate memory and fill the argument with a pointer to the newly allocated memory.)

Example:
```
getline  // only if arg 0 is a pointer to null
// (otherwise, no allocation done)
```

  - Any producer argument must be a pointer to a responsible pointer object.

- Irresponsible pointer arguments (A function that does not produce or consume a pointer argument effectively treats that argument as 'irresponsible'. It is permissible to pass any pointer (responsible or not) to such a function.)

Example:
```
strcpy() // both arguments irresponsible
```

- Forced-Irresponsible pointer arguments (A function that takes a pointer-to-pointer argument and modifies it, without modifying the memory or consuming it.)

  - A responsible pointer must not be passed as a forced-irresponsible pointer argument.

Example:
```
void fn(char** x) { // x points into a string
  (*x)++;
}
```

Because pointers can also be used in other C objects, we must extend our notions of responsibility to other objects. Here are the properties for structs and C++ classes:

- A struct is responsible if it contains any responsible pointers, or it contains any other responsible objects.
- A responsible object is good if all of its responsible pointers are good or null.

- A responsible object is a zombie if none of its responsible pointers are good. (Unlike pointers, objects can first exist in a zombie state when its responsible pointers are uninitialized).
- A responsible object is inconsistent if it is neither good nor zombie.
    - Any function that modifies the state of an object must not exit with the object in an inconsistent state.

Example:
```
typedef struct foo_s {
  char *rptr1;  // responsible
  char *rptr2;  // responsible
} foo_t;

void f1(foo_t* foo) { // assumed to be good
  free( rptr1);
} // bad, rptr2 still good, foo inconsistent

void f2(foo_t* foo) { // assumed to be good
  free( rptr1);
  free( rptr2);
} // ok
```

Here are the properties for unions:

- A union that contains a pointer may have that pointer be out-of-scope, irresponsible or responsible. (The rules for pointers in unions are the same as for structs. However, since all elements in a union share memory, reading or writing to a union member reads or writes to the pointer. A union with a responsible pointer thus has two extra rules.

Example:
```
#define SIZE (siezof(char*))

typedef struct foo_u {
  char *rptr;         // responsible
  char other[SIZE];   // any other data
} foo_t;
```

- Reading any element in a union with a responsible pointer in a good state converts the pointer to a zombie. (This assumes that the pointer's value was read and assigned elsewhere somehow)

Example:
```
char string[SIZE];
foo_t foo;
memcpy( &foo.string, foo.other, SIZE);
// foo.rptr: good -> zombie
```

- Writing any element in a union with a responsible pointer sets the pointer state to uninit (its pointer value is invalid)

Example:
```
foo_t foo;
foo.rptr = NULL;
foo.other[0] = 'a'; // foo.rptr is invalid
```

Here are the properties for arrays:

- An array of irresponsible pointers needs no monitoring (other than the standard rules wrt irresponsible pointers).
- An array of responsible pointers is like any other responsible object, but with one wrinkle: a size value. The size value indicates the number of responsible pointers in the array. It might be set to the index of the last element or one past the last element. It might also be an irresponsible pointer that points to the high element or one past the array.
- It is acceptable for this number to be less than the array bounds; the array may have several unused & uninitialized values beyond the size. (This means we do not prevent buffer overflow on the array.)
    - Any responsible array must have a 'size' variable declared in the same scope as the array that indicates the number of pointers to be used in the array.

Example:
```
void f() {
  int *array = malloc(10 * sizeof( int)); // responsible
  int size = 8;
  for (int i = 0; i < size; i++) {
    array[i] = malloc(sizeof( int));
  }
} // ok
```

Typecasts are easily handled by POM. There is no problem with typecasting a pointer (responsible or irresponsible) to a pointer of different type. Likewise, there is no issue with converting an irresponsible, or out-of-scope pointer to an integer or some other representation (such as a char array), or vice versa. A responsible pointer can be converted to an integer, this could be considered an implicit assignment to an irresponsible pointer and then conversion to int.

4.2 Producers and Consumers

A function that produces a responsible object (which may just be a responsible pointer) is a producer

function. Examples of producers include malloc(), calloc() and various special functions like strdup().

Some functions take as an argument a pointer to a pointer, if not NULL, they allocate space for an object and set the pointer to the produced space. (Example: getline() from GNU glibc.) In other words, a producer function can produce a pointer in one of its arguments, not just in its return value.

A function that frees a responsible object is a consumer function. The most well-known example of such functions is free().

It is possible for a function to consume multiple responsible objects. It is also possible for a function to produce one argument (or return value) and consume an argument. A function can produce an object via its return value, but it cannot consume an object via return value, only via arguments.

A producer function need not even invoke malloc(). For example, a producer function could split a responsible pointer from a responsible object and return the pointer. The following is a producer function because it returns a pointer that must be subsequently freed.

```
typedef struct foo_s {
  char *a;  // responsible
  char *b;  // irresponsible
} foo_t;

// producer function
char *function(struct foo_t* foo) {
  char *tmp = foo->a;
  foo->a = NULL;
  return tmp;
}
```

The following function consumes its second argument, because its second argument no longer needs to be explicitly freed; it is incorporated into a responsible object.

```
// consumer function
void function2(struct foo_t* foo, char *a) {
  if (foo->a != NULL) {
    // handle error, what to do with current foo->a?
  }
  foo->a = a;
}
```

The behavior of the realloc() function depends on its size argument. It requires a responsible pointer (that is good or null), so the pointer argument is consumed. It also returns the pointer successfully reallocated, or a pointer to the new internal copy (if it had to copy), or NULL if it failed to copy. So it also has a producer return value. Some implementations treat a call to realloc() with a size of 0 as a call to free(), but this is nonstandard behavior. The C11 standard declares that specifying 0 as the size is an obsolete feature, so we will ignore this case.

4.3 Analysis

For each function declaration, we need to know if the function returns a responsible pointer; that is, is it a producer? Also we need to know if the function produces any argument pointer-to-pointer. Finally we need to know if the function consumes any argument pointer.

We can use static analysis to answer these questions for a function whose definition is available. But if a function's definition is not available, we require some notation to answer these questions.

The main() function must not produce or consume any arguments.

To analyze a function body properly, we must inspect its definition. We must view all of the variables available to the function. That includes arguments, local variables, and static variables. Does the function produce or consume any of them?

A responsible pointer can be in multiple states at once. We always assume that the states a pointer is in can be determined statically. For any two states, it is easy to use branching to create a pointer that could be in both states. Consider this example:

```
char *rptr; /* uninitialized */
bool flag = /* ... */
if (flag) {
  rptr = new char[...];
}
/* rptr now good or uninitialized */
```

This can cause trouble later on; there is no way to distinguish good responsible pointers from uninitialized pointers. To be safe, this code must use the flag variable to determine if the pointer is good, and determining flag's value statically might be impossible.

We can handle this code statically by allowing rptr to be either good or initialized and indicate an error if either state causes an inconsistency later in the code. This means that this code will always yield an error: either when rptr is freed (because it may have been uninitialized) or when rptr leaks (it may have been

good). The code can always be brought to compliance with POM by initializing rptr to NULL:

```
char *rptr = NULL;
bool flag = /* ... */
if (flag) {
  rptr = new char[...];
}
...
free(rptr); // was good or NULL
```

Besides using control flow to create multiple states, malloc() & other producers can make pointers that are either good or null. No other combinations of states are typically possible without control flow. But because of control flow, we must assume any combination of states is possible. So a responsible pointer's state should be expressed as a list of booleans (or a bit-vector), one for each possible state.

However, this does mean that this safe code snippet does not abide by POM:

```
char *rptr; /* uninitialized */
bool flag = /* ... */
if (flag) {
  rptr = new char[...];
}
// ...
if (flag) {
  free( rptr);
}
```

4.4 Remaining Issues

POM has several unresolved issues. These can be tackled in future implementations using a waterfall model of development. That is, future designs for POM can tackle these issues, with only minor extensions to the current design:

- How should POM handle a circular linked list? By POM's model, either every pointer in the list is responsible, or every pointer is irresponsible. Irresponsible pointers renders the list useless for managing the elements, but responsible pointers would prevent any responsible pointer from pointing to the list from outside it.
  - One solution would be to have one pointer be explicitly irresponsible, but this would be difficult to enforce. If the list contains one special 'sentinel' element, then POM could simply declare that the pointer to the sentinel element is irresponsible while all other pointers are responsible.
- Can a responsible pointer be created from an integer or other non-pointer representation? An initial implementation can simply disallow this possibility; later we can investigate programs that require this capability.
- Such a list would not be considered compliant with POM, as every pointer is responsible, but no
- How should C++ code be handled in POM? (The current design ignores C++, focusing on C. C++ is more complex but no new fundamental approaches are required.)
- Is there any way to limit the lifetime of an irresponsible pointer? Irresponsible pointers that refer to freed memory are still an unsolved problem.
- Are there any functions that require an irresponsible non-constant pointer argument (because it is modified by pointer arithmetic, for example?) Should POM consider this capability?

Example:

```
void f(char** ptr) { // ptr must be irresponsible
  (*ptr)++;
}
```

- Would the model be damaged if the zombie and invalid responsible pointer states were merged into one?
- How should POM annotate functions that may duplicate a responsible pointer? One such function would be memcpy() if it is given memory which contains a pointer.

5. Prototype

A LENS-funded POM prototype has been developed as a proof-of-concept. The prototype is built using the C Intermediate Language (CIL), which is freely available at http://kerneis.github.com/cil/. CIL differs from the parsers used by typical compilers like GCC because it simplifies the C grammar. This makes syntax trees produced by CIL easier to analyze than trees built using traditional compiler front-ends, while simultaneously limiting its applicability. Eventually we will also build an implementation using mainstream static-analysis and compiler technology such as Clang/LLVM. This has the advantage over CIL of being able to analyze a broad code base, as well as supporting future extension of POM to handle C++ programs. For example, virtually all of the code in

Mac OS X 10.7 Lion and iOS 5 were built with Clang and llvm-gcc.

6. Risks and Mitigations

- *Risk*: Program uses different ownership model.
- *Mitigation*: Notation needs 'out-of-scope' indicator to not check pointers that use a different ownership model.
- *Risk*: Program uses nonstandard extensions to the C language, or relies on undefined or implementation-defined behavior
- *Mitigation*: Nonstandard extensions won't affect model, but will degrade accuracy of the implementation.
- *Risk*: Program contains memory management bugs in non-analyzable code (such as assembly code or nonstandard extensions).
- *Mitigation*: Notation needs a 'trust-me' indicator when the verifier cannot verify code safety. The developer must assume responsibility that any such code is secure.
- *Risk*: Program intentionally fails to free memory; most likely because its platform typically frees the memory when the program exits.
- *Mitigation*: Verifier needs an option to ignore memory leaks.
- *Risk*: Notation is too complex or cumbersome
- *Mitigation*:
    - Advisor should fill out model as much as possible by identifying responsible vs. irresponsible pointers. This minimizes developer involvement.
    - We need to measure notation complexity. (How much notation is required per 1000 lines of code?)
    - Need to measure learning curve, that is, how much effort must a developer use before they discover a memory bug?

7. Summary

The Pointer Ownership Model can be used as a static analysis tool to verify the memory safety of C (and eventually C++) programs. It is based on a general strategy that states that judicious use of semantics added by a developer can greatly aid static analysis tools. Unaided, static analysis runs in to many limitations, such as the inability to recognize null pointer dereferences with a high accuracy. We believe that requiring developers to provide a small amount of carefully-chosen semantic information can be a great aid to static analysis, and POM is an example of this strategy, applied to dynamic memory management.

Proper memory management is currently obtainable using various techniques, such as garbage collection or reference counting. All such strategies impose a performance penalty, and none are provided with most implementations of C. In contrast, C++ offers reference counting via the shared_ptr<> template, and many newer languages require the developer to use some high-level form of memory management. POM imposes no performance overhead at run-time, and only a minor penalty at compile-time to validate pointer usage.

POM is designed to work with existing C code, and one of our goals is to demonstrate its effectiveness by using it on a large open-source program. It may require some additional annotations from a developer, but it requires no code changes (unless it detects memory bugs). Consequently, the cost for developers to test a program using POM should be minimal.

8. References

[1] Thomas Plum and David Keaton, "Eliminating Buffer Overflows, Using the Compiler or a Standalone Tool", Workshop on Software Security Assurance Tools, Techniques, and Metrics (SSATTM), Long Beach CA Nov 7-8 2005. http://www.plumhall.com/ASE-SSATTM-plum+keaton-proceedings.pdf

[2] R. B. Dannenberg, W. Dormann, D. Keaton, R. C. Seacord, D. Svoboda, A. Volkovitsky, T. Wilson, and T. Plum. "As-If Infinitely Ranged Integer Model." In Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE '10), Washington, DC, pp. 91-100. Los Alamitos, CA: IEEE Computer Society, 2010.

---

[i] [