A Pointer Ownership Model for C Inspired by Rust

Muhammad Hassnain (mhassnain@ucdavis.edu)

Temporal Memory Safety

- Accessing Memory at the wrong time
- Use after free/double free etc.

```
int main()
{
    int *p = new int;
    *p = 42;

    free(p);

    cout << *p << endl;
    return 0;

}
```

Contributions

- Formal Ownership and Borrowing Model for C that tracks:
  - Responsibility (Ownership)
  - Mutability
  - State (Good, Null, Zombie)
- A borrowing and lifetime analysis that enforces single writer, multiple reader on pointer aliases.
- An LLM assisted Program Ownership Model (POM)
- Evaluation on 5 CWEs and 100% unsafe detection, and 90% safe code identified correctly!

Model

Aim to fix bugs in three stages:

1. A Mental model
2. Automated generation of p-model
3. POM verifier

Classification of Pointers

Divide pointers into three types:

1. **Responsible:** Manages the pointer and must free it.
2. **Irresponsible:** No involvement in allocating and freeing.
3. **Out of Scope:** No constraints on these.

Responsible Pointers

Shepherd heap memory and ensures the memory eventually gets freed.

Responsible pointers are further classified into three states (move semantics):

1. GOOD
2. NULL
3. ZOMBIE

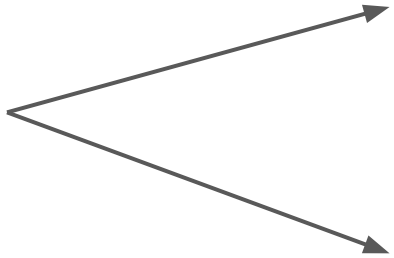
1. GOOD Responsible Pointers

They are sole responsible pointer to initialized memory (no aliasing).

It's a violation of POM for two GOOD responsible pointers to point to same heap object.

```
int * p = new int;
int *q  = p;
```

If P is a good responsible pointer



q is an irresponsible pointer.

p has delegated ownership and is now a ZOMBIE

2. ZOMBIE Responsible Pointers

This can be uninitialized, freed or pointing to memory managed by some other responsible pointer.

Dereferencing a ZOMBIE pointer violates the POM design.

Responsible Pointer Example

```
3 // ...
4 void usage(char* msg) {
5   fprintf(stderr, msg);
6   free(msg);
7 }
8
9 int main(int argc, char**
      argv) {
10    char* errmsg;
11    if (argc <= 2) {
12      errmsg = malloc(100);
13      if (errmsg != NULL) {
14        snprintf(errmsg, 100,
      "Need more than %d
      arguments!", argc);
15        usage(errmsg);
16        free(errmsg);
17        exit(1);
18      }
19    }
20    // ...
21  }
```

Figure 1. Example C source code with a double-free bug

```
Functions:
  usage:
    args:
      msg:
        resp: responsible
        start: [GOOD]
        end: [ZOMBIE]
    return: []

  main:
    args:
      argv:
        resp: diligent
        start: [VALID]
        referent:
          resp: diligent
    locals:
      errmsg:
        resp: responsible
```

Figure 2. Example p-model for code in Fig. 1

Irresponsible Pointers

Not responsible for freeing the memory they point to.

Can do pointer arithmetic and can also point to value from &.

Irresponsible pointer cannot be assigned the return value of a function that returns a responsible pointer.

Irresponsible Pointers & Lifetime

What if an irresponsible pointer outlives the memory it points to?

MINIMUM Lifetime: Minimum lifetime of whatever object pointer references. Anything this pointer points to must outlive this minimum lifetime.

Function Arguments

Two specific type of irresponsible function arguments:

1. Diligent:
2. Producer

1. Diligent Irresponsible Pointer Argument

Satisfies the following properties:

1. It never escapes the function body except as a return value.
2. The contents of the pointer may be changed.
3. But the contents will not be freed during its lifetime.

This helps in eliminating analysis of pointers that treat their data as read-only. This is like Rust reference.

2. Producer Irresponsible Pointer Argument

Satisfies the following properties:

1. Pointer to pointer argument.
2. Outer is irresponsible, but inner pointer is responsible.
3. It should be the only pointer with mut access to pointed memory.

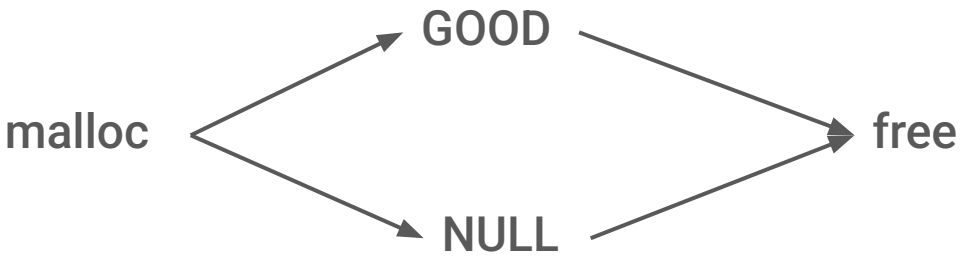
This is like Rust &mut.

Responsible Pointers as Function Arguments

A function argument pointer should be labeled responsible only if it consumes the argument.

A function that takes a responsible pointer that starts as GOOD and becomes a ZOMBIE is said to have consumed the pointer.

Control Flow and Responsible Pointer States



POM requires developer to ensure only valid pointers are passed to library functions.

The POM keep tracks of pointer states and will issue a warning if we dereference a pointer that is NULL or dereferenced.

Model Properties

Skipping - More formal view of above mentioned things.

Borrows and Lifetime

1. **Borrow Tracking**
   - When a pointer is copied from source to destination, POM records a borrow.
   - Borrows are recorded transitively and propagate to deeper dereference levels.
   - Tool tracks both may-alias and must-alias.

Borrows and Lifetime

2. **Borrow Constraints**
   - Pointer may not be read through it, if it has any outstanding mut borrows.
   - Pointer may not be written through or freed, if it has any borrows.
   - Borrowing in another chains, make both dest pointer immut.

Multiple direct borrows	Borrow chain

`p2 = p1;`
`p3 = p1;`
`*p3 = 42; /* violation */`
	`p2 = p1;`
	`p3 = p2;`
	`*p3 = 42; /* allowed */`

Borrows and Lifetime

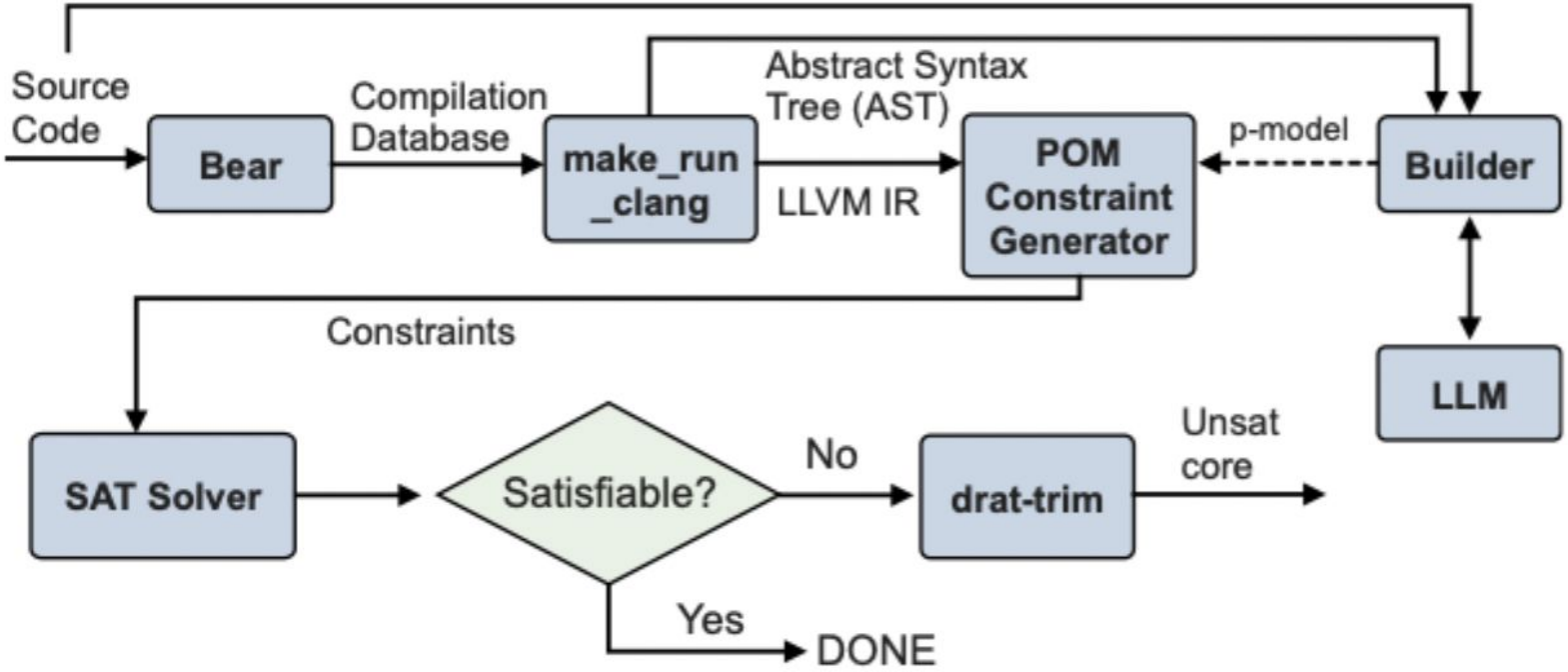
3. **Lifetime Analysis**: Uses Non-lexical lifetimes.

SAT Solving & Diagnostics

- Create a CNF. (Each clause is an OR of literals)
- Uses CryptoMiniSat.
- If satisfiable → valid p-model and program compiles with it.
- If unsat → get the falsifying, contradictory example and use LLM to explain it.

Implementation

103 last para?



Evaluation

1. Test Cases
    - Juliet C/C++ CWE examples.
    - Supports 4144/4604 test cases.

Evaluation

I don’t understand this :(



Figure 4. LLM builder, then SAT verifier: Juliet test-case outcomes for the five temporal-memory-safety CWEs. Successful GOOD and Successful BAD denote cases for which the tool correctly accepts GOOD examples or rejects BAD examples, respectively.