Carnegie Mellon University
Software Engineering Institute

Design of Enhanced Pointer Ownership Model for C

David Svoboda
Lori Flynn
Will Klieber
Ruben Martins
Sasank Venkata Vishnubhatla
Nicholas H. Reimer

September 2025

TECHNICAL REPORT
CMU/SEI-2025-TR-008
DOI: 10.1184/R1/29971765

CERT Division

http://www.sei.cmu.edu

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM25-1179

Table of Contents

Abstract

This report describes the design for a new temporal memory safety model for C code and an implementation to enforce it. The design improves on the CERT Pointer Ownership Model with enhancements including the use of large language models to complete a per-program model; an improved mechanism to prevent use-after-free errors, inspired by Rust's borrow checker and object lifetimes; improved function argument handling with a new abstraction of diligent or producer arguments; handling structs, unions, or arrays that contain pointers; and correct handling of ambiguity in assignment operations. This report details the research approach and early stage results of designing this model, its extension to C's type system, the tool design methodology, and the design and initial engineering of lightweight specification and validation tools.

1 Introduction

Memory-related bugs, such as use-after-free and buffer overflows, are among the most prevalent and challenging issues in C and C++ code. These bugs are difficult to detect and fix, often resulting in significant security vulnerabilities and system instability. Formal methods, which use rigorous mathematical and logical reasoning, offer a powerful approach to systematically identify and eliminate these types of bugs, providing greater confidence in software reliability and safety. Formal methods are traditionally labor intensive and difficult to scale, with ongoing research making inroads against those challenges [2, 11, 20, 24, 32, 33, 36, 43]. Lightweight formal methods—our work's category—balance rigor with usability, focusing on targeted partial analysis and the verification of systems, often using automated tools and model checking [47].

This report describes the design of the Enhanced Pointer Ownership Model (EPOM), a new temporal memory-safety model for C code, together with an approach to automatically check whether a program satisfies this model. It specifies our research approach and early results of designing this model, its extension to C's type system, our tool design methodology, and our design and initial engineering of lightweight specification and validation tools. Our design improves on the CERT Pointer Ownership Model [41] with enhancements that include use of a large language model (LLM) to complete a per-program model; an improved mechanism to prevent use-after-free errors, inspired by Rust's borrow checker and object lifetimes; improved function argument handling with a new abstraction of diligent or producer arguments; handling structs, unions, or arrays that contain pointers; and correct handling of ambiguity in function calls.

The heap is a part of a computer's memory that is used during a program's execution for dynamic memory allocation and deallocation of memory blocks of varying size and varying lifetimes. To avoid heap errors, C and C++ developers must be disciplined in preserving temporal memory safety in the heap. Such discipline requires them to internalize a mental model of how the platform will manage heap memory as instructed by their code. This model can be informed by literature, such as CERT Recommendation MEM00-C, which specifies to allocate and free memory in the same module, at the same level of abstraction [40]. Technology can also provide constrained models, which we will call "formal models," such as EPOM and Resource Acquisition Is Initialization (see Section 4). We will refer to the contents of files that specify the application of a particular formal model to an actual program as a "p-model." Many formal models of C memory and pointers exist, along with verifiers for memory safety, as detailed in Section 4.

EPOM is a way to enforce temporal memory safety. We are designing EPOM with the goal of being intuitive for C programmers who work with dynamic memory, since that work requires a mental discipline very similar to what EPOM requires. We have developed a conceptual model, and our team is now developing a builder and a verifier for it.

EPOM is not designed to verify that every valid C program is temporally memory-safe. Some programs, even well-defined ones, will violate EPOM to preserve scalability due to the model's limitations. This simply means that you cannot use EPOM alone to completely confirm a C program's memory safety; you must use an additional mechanism. For example, EPOM does not handle (1) reference-counted smart pointers, such as C++'s `std::shared_ptr<>`;(2) cyclic pointers, such as in doubly linked lists; (3) Windows handles; and (4) pointers managed by some other mechanism, such as a garbage collector.

However, EPOM can provide strong memory guarantees: *If all pointers in a program conform to EPOM, then the program is temporally memory-safe.* The EPOM model is also strong

enough to capture a wide range of memory vulnerabilities. For example, using the Common Weakness Enumeration (CWE), EPOM addresses CWE-401: Missing Release of Memory After Effective Lifetime or memory leak [25], CWE-415: Double Free [26], CWE-416: Use After Free (UAF) [27], CWE-476: NULL Pointer Dereference [28], CWE-590: Free of Memory Not on Heap [29], and some of CWE-908: Use of Uninitialized Resource [30].

2 Enhanced Pointer Ownership Model (EPOM)

EPOM is designed to help developers avoid, identify, and fix these temporal memory-safety issues in three stages:

1. Following the EPOM mental model should help developers avoid writing code with certain kinds of temporal memory-safety bugs.
2. Automated generation of the p-model identifies some errors for code where the developer forgot to comply with the model.
3. The EPOM verifier identifies all remaining EPOM compliance errors.

The EPOM builder and verifier are designed to assume that every pointer is either "responsible," "irresponsible," or "out of scope." If the p-model's label of the pointer's responsibility does not agree with the pointer's usage in code, then that constitutes an EPOM violation and the verifier should catch it. The user should investigate each error. If the user decides that the pointer is out of scope (i.e., it is managed by some other mechanism), then they should add this information to the p-model. The EPOM builder and verifier ignore pointers labeled as out of scope.

We use the term *heap object* to denote any single data structure whose memory is allocated with `malloc()`, `calloc()`, `aligned_alloc()`, or `realloc()`. Objects not allocated using one of these functions are not heap objects even if they live on the heap on some platforms. In EPOM, C pointers are partitioned into responsible pointers, irresponsible pointers, and out-of-scope pointers. Each responsible pointer stewards the object it points to; responsible pointers are intended to point to something on the heap that must be freed. Irresponsible pointers have no involvement with allocating or freeing memory, but they may point to anything (including on the stack or data segment or into objects), and irresponsible pointers may undergo pointer arithmetic. EPOM enforces no constraints on out-of-scope pointers. Their memory safety should be checked by some other mechanism. However, an out-of-scope pointer may not be assigned the value of a responsible pointer, and a responsible pointer may not be assigned the value of an out-of-scope pointer.

The terms *responsible*, *irresponsible*, and *out of scope* can be treated like type qualifiers in C (e.g., `const` or `restrict`). They subtype the pointer variables, irrespective of the variables' values. As with types, these qualifiers apply to a variable throughout its lifetime. For example, if `p` is considered to be a responsible pointer, it remains responsible throughout its lifetime and cannot cease to be responsible. These terms can apply to local pointers, pointers defined in structs or unions, pointers defined as function arguments, and the return value of a function if it is a pointer type. They can also apply to static pointers, but EPOM doesn't support static pointers yet.

2.1 Responsible Pointers

Responsible pointers are the subset of pointers that shepherd heap memory and ensure that the memory eventually gets freed. Each chunk of heap memory (i.e., a heap object) may be accessed directly at most by one GOOD responsible pointer. Responsible pointers never point inside the stack, inside the data segment of memory, or inside a heap object except to its beginning.

In ISO standard C, the value of a pointer can be in one of three states: INVALID (i.e., uninitialized), NULL, or VALID (i.e., pointing to initialized memory) [17]. Like ISO C pointers,

responsible pointers have states that can change over their lifetimes. The states for responsible pointers are ZOMBIE (uninitialized or pointing to memory managed by another responsible pointer), NULL, and GOOD (being the sole GOOD responsible pointer to initialized memory). GOOD pointers in EPOM are a subset of VALID pointers in ISO C, and ZOMBIE pointers are a superset of INVALID pointers.

It is a violation of EPOM for two GOOD responsible pointers to point to the same heap object. Therefore, assigning one responsible pointer to another and passing a responsible pointer as a function argument are problematic. Both involve copying the address value from one pointer to another. In EPOM, any assignment expression whose right-hand side indicates a responsible pointer can have two interpretations: The left-hand side could indicate an irresponsible pointer; see Section 2.2: Irresponsible Pointers for details. Or the left-hand side could indicate a responsible pointer, in which case the right-hand pointer is no longer considered GOOD. While it remains responsible, it has delegated ownership of the pointed-to object to the assigned-to pointer and is now a ZOMBIE. The states of responsible pointers in EPOM thus employ move semantics [4]. Dereferencing a ZOMBIE pointer violates EPOM, even if the pointer's value remains VALID.

Likewise, passing a responsible pointer to a function provides the same ambiguity that assignment does. If the function expects an irresponsible pointer as the argument, this is like assigning a responsible pointer to an irresponsible pointer. If the function expects a responsible pointer as the argument, then the function argument receives the same state as the original pointer, and the original pointer becomes a ZOMBIE: It has delegated ownership of the pointed-to object to the function argument.

Responsible pointers avoid the aliasing problem [38] by enforcing (at compile time) that for each object on the heap there is exactly one GOOD responsible pointer that points to it.

2.2 Irresponsible Pointers

Irresponsible pointers are not responsible for cleaning up the memory they point to. Since they do not participate in memory allocation or deallocation, the main concern with irresponsible pointers is that they must respect temporal memory safety. CERT POM had irresponsible pointers but used no tracking mechanism akin to lifetimes, so it did not prevent use-after-free errors. EPOM does.

An irresponsible pointer cannot be assigned the return value of a function that returns a responsible pointer (such as `malloc()`). Unlike a responsible pointer, an irresponsible pointer can be assigned a value resulting from pointer arithmetic or a value created by C's address-of operator `&`.

The only liability for an irresponsible pointer is that it might outlast the object it points to, which might cause a use-after-free error. Due to the aliasing problem [38], we do not expect to track the value of every irresponsible pointer. Instead, we track the lifetime of every object that an irresponsible pointer can reference. This is based on Rust's borrow checker and lifetimes. When we speak of an irresponsible pointer's lifetime, we refer to the minimum lifetime of whatever object the pointer references. The pointer may be assigned to a different object, but that object must still outlast the minimum lifetime. (In fact, every object has a lifetime; it is only irresponsible pointers whose objects' lifetimes must be explicitly tracked.)

By default, irresponsible pointers that are function arguments are assumed to outlast the function call; their lifetimes need not be specified unless their referenced object does not outlast the function call. The same applies to locally declared irresponsible pointers. Irresponsible pointers that serve as function return values are different; they must outlast the function call. Therefore, they must provide an explicit lifetime, which will typically match the lifetime

of one of the function arguments. Or they could specify the lifetime "STATIC," meaning that their object lives until the program terminates.

Irresponsible pointers affect other types that contain them, such as arrays, structs, and unions. Any composite type that can contain an irresponsible pointer should have a lifetime specified; this specification can be used to indicate the lifetime of the irresponsible pointers contained in the composite type.

2.3 Function Pointer Arguments

EPOM improves on POM's handling of function arguments that are pointers. A pointer that is passed into a function could be responsible, irresponsible, or out of scope. While the pointer's responsibility never changes during the function's execution, the pointer's state may change. Thus, every pointer argument has an initial set of states and a final set of states. These can be identical but need not be. There are five responsibility classifications of function arguments of pointer type: out of scope, diligent, irresponsible, responsible, or producer. The notion of diligent or producer arguments is new to EPOM; they did not exist in POM.

Out-of-Scope Arguments

Out-of-scope arguments are outside our model.

Diligent Arguments

Diligent arguments are pointers that could be irresponsible or responsible. In particular, during function execution, they are never assigned to point somewhere else, they are never freed, and their value never escapes the function. They are never assigned to another pointer with one exception: The function may return their value as an irresponsible pointer.

Diligent pointer arguments have several constraints: Any pointer may be passed to a function that expects a diligent pointer argument. The pointer's start state should be noted, but not its end state, because its end state will always match its start state. It is permissible to pass any pointer (responsible, irresponsible, or out of scope) to such a function. For example, `strcpy()` takes two diligent pointers and returns its destination string pointer. Passing it any pointers, as long as they point to VALID memory, complies with EPOM. The return value's lifetime matches the first argument's lifetime (since it is the first argument).

Irresponsible Arguments

An argument whose value is never freed, but might be assigned to point elsewhere or have its pointer reassigned, should be irresponsible. These pointers' start states and end states should be specified in the p-model.

Responsible Arguments

A function that takes a responsible pointer that starts as GOOD but may become ZOMBIE is said to "consume" the argument. Consequently, a function argument pointer should be declared "responsible" only if the function might consume the argument; that is, it might free the pointer or assign it to another responsible pointer. For instance, the `free()` function converts a responsible pointer (either GOOD or NULL) into a ZOMBIE. As with irresponsible arguments, responsible pointers' start states and end states should be specified in the p-model.

A function with one or more responsible pointer arguments may change the arguments' states distinctly based on whether it produced an error or not. For example, the `realloc()` function takes a responsible pointer as its first argument. If the pointer was NULL, it remains

NULL. If it was GOOD, and `realloc()` returns a non-NULL pointer, it becomes a ZOMBIE. If `realloc()` returns NULL, the pointer's state remains unchanged. So this argument pointer is similar to `free()`'s argument pointer, but only when `realloc()` succeeds. But `realloc()` also returns a responsible pointer, which is either GOOD or NULL like `malloc()`.

Producer Arguments

A "function producer argument" is a pointer-to-pointer argument with special properties. The outer pointer (i.e., the argument in the function invocation) must be constant; that is, the function may not assign it to point elsewhere. The outer pointer may not be freed or consumed. Finally, any call to the function must provide either a responsible outer pointer or the address (`&`) of a pointer.

In these cases, the outer pointer is treated as responsible, except that it may not be consumed or freed. The inner pointer may be responsible, irresponsible, or out of scope. They are similar to diligent arguments, but diligent arguments are expected never to be consumed or freed. Their pointers and pointed-to values are considered "read-only."

A function producer argument that points to responsible pointers may free or otherwise consume them. Producer arguments are neither responsible nor irresponsible due to the above constraints. If the outer pointer was responsible, it would be possible to free it, and if the outer pointer was irresponsible, it would be possible to use it to circumvent the responsible pointer aliasing. A function with one or more producer arguments may change the arguments' states based on whether it produced an error or not.

2.4 Types that Contain Pointers

EPOM has a design for types that contain pointers, which POM did not handle. We define a *composite type* as any C data type that can contain a pointer. Composite types consist of pointer types and structs, unions, arrays, and pointers that contain a composite type. We distinguish these from *non-composite type*s, which include structs, unions, and arrays that do not contain any pointers. A composite object is an object of a composite type. A responsible composite object is a composite object with at least one responsible pointer, and an irresponsible composite object is a composite object with at least one irresponsible pointer. Note that a composite object can be both responsible and irresponsible.

A responsible composite object with exactly one responsible pointer has the same responsible states as the pointer. That is, if the responsible pointer is GOOD, the composite object's responsible state can be inferred as GOOD. A composite object with more than one responsible pointer will also have a responsible state derived from the responsible pointers' states. Likewise, a composite object with exactly one irresponsible pointer itself can have the same states as the pointer. That is, if the irresponsible pointer is VALID, the composite object's irresponsible state can be inferred as VALID. A composite object with more than one irresponsible pointer will also have an irresponsible state derived from the responsible pointers' states.

In C, many heap objects are not accessible directly via a pointer defined on the stack but can be accessed indirectly through two or more pointers. An example is the third element in a linked list.

We define a *C-path* as a way to access any object in memory in C. It starts off with a global or local variable and then consists of a (possibly empty) sequence of array accesses (e.g., `a[i]`), pointer dereferences (e.g., `*p`), struct membership (e.g., `s.a`), and union membership (e.g., `u.a`). C-paths are a lot like file paths. Composite types are what one uses to build networks of heap objects in memory. The pointers must be VALID or GOOD.

A heap object that can't be referenced by any C-path indicates a memory leak. If no memory

leaks exist in a program at a point in time, then every heap object has at least one C-path to reference it. In a memory-safe program with no out-of-scope pointers, at any point during program execution, every heap object has exactly one C-path where every pointer in the path is responsible. We call this "the responsible C-path." It is an EPOM violation to free a pointer via a C-path that has at least one irresponsible pointer in it. Note that any variables before the first pointer live on the stack or global segment, and everything past the first pointer must live on the heap.

2.5 Control Flow and Responsible Pointer States

A 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, branching can create a pointer that could be in both states. For example, `malloc()` returns a responsible pointer that could be GOOD or NULL.

This can be a source of trouble. In Standard C, there is no way to distinguish GOOD responsible pointers from uninitialized pointers. This (among other things) requires a developer to maintain internal discipline to make sure that only VALID pointers are passed to most library functions. EPOM is designed to keep track of the states of pointers and issue warnings. For example, the EPOM verifier will warn if a pointer that might be uninitialized or NULL is dereferenced.

3 Implementation

Each p-model is stored in a YAML file. Figure 3.1 shows an example of C source code, and Figure 3.2 shows its associated p-model. See Appendix A for our YAML schema for p-models. We will also build a verifier to confirm that a p-model correctly models the responsibility of all pointers in the code. To verify a p-model and source code, you must first use Clang on the source code to generate an Abstract Syntax Tree (AST), then serialize the AST to a JavaScript Object Notation (JSON) file. EPOM verification by design works only on programs that can be compiled with Clang. This design enables the AST to be inspected without having to integrate *our* code with Clang. Appendix B provides the details of our planned implementation and includes a high-level flow diagram of verifying a p-model in Fig. B.1.

Once the AST is serialized in JSON format, the verifier can ingest the AST to track and build an internal pointer model of the pointers in the program. A simple dictionary is used to map functions found in the AST to the function's internal control flow, argument pointers, local variable pointers, and return type pointers. Given the tree structure of the AST JSON, each function definition will contain all necessary information to build the internal pointer model. Based on the AST node type, the internal pointer model will digest the AST node accordingly and update the associated function's internal control flow if necessary.

After the internal pointer model has been fully created from the digested AST JSON for each function, the p-model is compared to the end state of the internal pointer model after following the internal control flow. First, the p-model checks for the existence of all declared function argument pointers, local variable pointers, and the return pointer type. If there are any missing or extraneous pointers, the verifier will warn the user of the discrepancy. Afterward, the function argument pointers and return pointer type in the p-model are verified for correctness given the internal control flow. Once the function argument pointers and return pointer type in the p-model are verified, the local variable pointers are verified for correctness given the internal control flow. Any verification errors are reported back to the user as warnings.

Our p-models are currently built manually, but we will automate that. Automated static analysis will help but gets complicated. For example, determining the responsibility of a pointer inside a struct requires inspecting how the struct is used throughout the program. So we will use an LLM to help complete automated p-model generation (CERT POM instead required manual completion). We hypothesize that an LLM may be able to correctly ascertain the

```
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) {
1     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 3.1: Example C Source Code

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

  main:
    args:
      argv:
        type: array
        resp: diligent
        max: argc
        start: [VALID]
        referent: diligent
    locals:
      errmsg:
        resp: responsible
```

Figure 3.2: Example p-model for the Code in Figure 3.1

responsibility of some pointers that static analysis alone may not resolve correctly and do it faster and more accurately (i.e., with a greater percentage of correct labels in the p-model) than a human could. Manually creating or verifying a p-model is slow and impedes its use; for example, the specifications used by Frama-C's library must be proofread by the user [38]. An EPOM p-model that can be generated automatically does not have this impediment. We also hypothesize that an LLM may be better at discerning programmer intent than static analysis alone, especially if the code is defective or violates EPOM. A risk of using LLMs is that they sometimes hallucinate, making wrong statements, often in confident language. However, since the verifier will assess the accuracy of a p-model, it will emit warnings on any p-model that the program does not comply with, therefore preventing any hallucinations from producing a "correct" p-model. We plan to test how successful the LLM will be in filling out the p-model. Since a p-model can be filled out manually, it is easy to "grade" the LLM, and its performance will be a major component of this research. As we continue to develop EPOM based on this design and then test it, we want to investigate which LLMs perform best and how to optimize LLM prompts to output good p-models.

4 Related Work

Memory violations are among the most prevalent and severe types of vulnerabilities, and they have been extensively studied by researchers in academia and industry for decades [42]. Memory-safety violations primarily fall into two categories: temporal and spatial. However, Chen and colleagues have identified other important classes [10, 9, 8], including invalid pointer value errors (such as null, uninitialized, or manufactured pointers), memory leaks, and segment confusion errors (such as invalid dereferences and invalid frees). In this work, we focus on temporal memory safety by tracking pointer ownership and checking compliance with temporal safety constraints.

Several prior formal models have addressed temporal memory safety, with our approach specifically inspired by three main directions [12, 18, 41]. First, the Pointer Ownership Model (POM) [41] allows developers to formally specify temporal memory-safety constraints through a p-model, which can be partially inferred using static analysis and validated automatically. Our work extends POM by enhancing its memory-safety coverage through explicitly tracking pointer lifetimes.

Second, our approach is influenced by Rust's borrow checker, which ensures comprehensive memory safety, including thread safety and controlled concurrency [18]. Our approach supports a limited set of Rust features, specifically those similar to Rust's Box pointers and references, but does not cover Rust's reference-counted pointers. An alternative strategy explored in the Defense Advanced Research Project (DARPA) Translating All C to Rust (TRACTOR) program [13] involves automatically translating existing C code into Rust to eliminate memory-safety vulnerabilities. However, such translations may use unsafe Rust code, thereby undermining the safety guarantees that Rust provides.

Third, our approach aligns conceptually with the Resource Acquisition Is Initialization (RAII) paradigm from C++ [5, 12]. RAII systematically associates resource lifetimes—such as dynamically allocated memory, file handles, and mutex locks—with object lifetimes, encapsulating resource acquisition in constructors and resource release in destructors. Our method similarly tracks and manages pointer lifetimes statically, which allows us to check for temporal memory safety.

Other approaches to guarantee memory safety in C include extending the language with memory-safe pointers [34, 48, 49]. For instance, Checked C [14] extends C by introducing checked pointers that explicitly specify bounds for objects and arrays, allowing compile-time verification or run-time checks. Even though some semi-automated approaches have been suggested to transpile C into Checked C code [22], this method still requires rewriting large parts of the code base into a safe language, which limits its practicality. The MSA tool helps port C code to Checked C [31], and like our EPOM p-model builder and verifier, it uses compiler-generated ASTs, static analysis, and LLMs to deal with complexity. MSA also uses symbolic inference, unlike our system.

Alternatively, C-Rusted [3] incorporates Rust-inspired concepts like ownership and borrowing through standard C macros that vanish after preprocessing, enabling static analysis without altering compiler behavior. Its analyzer detects errors such as NULL pointer dereferences, memory leaks, and use-after-free, but extensive manual annotations limit scalability for large legacy codebases.

Another approach is to extend pointer structures to include bounds information, inserting run-time checks during compilation to ensure memory accesses remain within allocated regions [19, 39]. While effective, this modification introduces significant run-time and memory overhead [19].

Other approaches to temporal memory safety build on the low-overhead Capability Hardware Enhanced RISC Instructions (CHERI) architecture [15, 16, 45], which provides spatial memory safety and has ongoing standardization efforts with significant industry involvement [23, 44]. CHERI replaces the use of integer addresses as pointers with capabilities that carry bounds information, and hardware enforces its capability-authorized memory access. Although the temporal memory enhancements demonstrated efficient execution time overhead, these approaches are not available on mainstream CPUs (e.g., Intel x86-64 and AMD64) since CHERI requires capability registers, tagged memory, or hardware-enforced bounds checks for pointers.

Finally, model checking techniques [6, 7, 21, 35, 37, 46] represent a general approach for verifying memory properties but face scalability challenges due to their precise modeling of program memory. In contrast, our approach, although targeting only a subset of memory-safety properties, aims to be lightweight, scalable, and efficient.

5 Conclusion and Future Work

This report describes our research approach to demonstrate that our EPOM formal model can be useful to prove partial temporal memory safety in C code. We provide highlights and some details of our design for the EPOM model, C code evaluation, tool design methodology, validation tool, and testing ideas. We are now developing code for automating p-code creation and validation as well as an automated testing framework to run experiments. Our design choice to use an LLM for p-model generation support is intended to lower manual effort and increase correctness, but it risks hallucinations. We expect that such hallucinations will cause the verifier to produce warnings about the program not complying with the p-model. Planned tests will help us understand the impact of that and other design choices. EPOM is intended to help developers avoid, identify, and fix temporal memory-safety issues with its mental model, automated p-model generation, and automated verification. If successful at validating temporal safety, EPOM could improve the security and functionality of much of the huge amount of C code currently in use at a low cost (due to full automation) and with no performance reduction.

Future work could investigate how out-of-scope pointers interact with responsible and irresponsible pointers in data structures such as doubly-linked lists and reference-counted pointers and then possibly extend EPOM to include such data structures. Future work could increase C language coverage in EPOM by (1) supporting the `alloca()` function, which would require modifying the C-path definition; (2) supporting static pointers; and (3) supporting tracing responsible or irresponsible pointers through integer casts or casts to any other non-pointer types.

A Appendix A: YAML Schema for p-models

```
# YAML schema for p-models, compatible with Yamale
# To use this, run: yamale -s pom.yamale.yml <your pom file>
# Or, use the built-in checker command: pom-lint <your pom file>

Functions: map(include('function'), key=str(), min=0, required=True)

---
function:
  args: map(include('pom-element'), key=str(), min=0, required=False)
  locals: map(include('pom-element'), key=str(), min=0, required=False)
  return: include('recursive-pom-element', required=False)


pom-element:
  resp: include('responsibility', required=True)
  type: include('pointer-type', required=False)
  start: include('pointer-state', required=False, min=1)
  end: include('pointer-state', required=False, min=1)
  destructor: include('destructor', required=False)

  # conditional properties
  min: any(int(), str(), required=False)
  max: any(int(), str(), required=False)
  lifetime: any(int(), str(), required=False)

  referent: any(include('recursive-pom-element'), include('responsibility'), required=False)

recursive-pom-element:
  name: str(required=False)
  resp: include('responsibility', required=False)
  type: include('pointer-type', required=False)
  start: include('pointer-state', required=False, min=1)
  end: include('pointer-state', required=False, min=1)
  destructor: include('destructor', required=False)

  # conditional properties
  min: any(int(), str(), required=False)
  max: any(int(), str(), required=False)
  lifetime: any(int(), str(), required=False)

  referent: any(include('recursive-pom-element'), include('responsibility'), required=False)

responsibility: enum('responsible', 'irresponsible', 'out-of-scope', 'diligent', 'producer')
pointer-state: list(enum('VALID', 'INVALID', 'GOOD', 'ZOMBIE', 'NUL'), min=1)
pointer-type: enum('pointer', 'array', 'struct', 'union')
destructor: regex('^(free|fclose|.*)$')
```

Figure A.1: YAML Schema for p-models, Compatible with Yamale

B Appendix B: Details of Implementation

To create a p-model from C source, you must run Clang on the source code to generate an Abstract Syntax Tree (AST) and then serialize the AST to a file in JSON format. Our EPOM tooling currently works only on programs that can be compiled with Clang. From the AST, a Python `FunctionDigest` class can be created to represent each FunctionDecl. Given the tree structure of the AST JSON, all information about function declaration, parameters, and control flow is stored within the FunctionDecl object, and therefore within a `FunctionDigest` class.

The `FunctionDigest` contains information about the parameters (stored in a `ParameterDigest`), return type (stored in a `ReturnDigest`), control flow (stored in a `ControlFlowSequence`), and pointer ownership model status (stored in a `PointerOwnershipStatus`). The `PointerOwnershipStatus` lists out each parameter pointer and local variable pointer as well as the associated pointer responsibility and state. Based on the responsibility, a lifetime of the pointer is also stored within the `PointerOwnershipStatus`.

To create a `FunctionDigest`, a pointer state map is initialized. Then, the p-model builder follows the `ControlFlowSequence` linearly, making changes to the pointer state map based on the operations occurring in the control flow. Once it has fully digested the control flow, the p-model builder converts the pointer state map into a `PointerOwnershipStatus` object and stores it within the `FunctionDigest`. Figure 3.2 shows an example.

Based on the results of the pointer state map, the responsibility and state of some pointers may not be fully determined. In this case, an LLM is queried. We developed a prompt to query the LLM with necessary definitions for pointer ownership, the pointer state map, and the clang AST. The required output is an updated pointer state map with any unresolved responsibilities and states remedied by the LLM. Additionally, for each change, a reason is required to presented to the user.

See Figure B.1 for a high-level verification flow.

For verification, the clang AST of the C source that the EPOM is describing is necessary. To verify the YAML representation of a program's pointers, the data format must first be validated. Using the Yamale [1] library, the YAML representation is validated against a schema for p-model files (see Figure A.1). If validation does not succeed, then verification fails. Assuming validation is successful, the clang AST is converted into a map of function names pointing to `FunctionDigest`. Then, for every function within the AST, the YAML representation is queried for that function. If that function exists, then the arguments, local variables, and return type are all validated. If the function does not exist, a warning is returned to the verifier, but verification will continue.

The first step in verifying function arguments and local variables is the same: The YAML representation is queried to ensure that the proper function arguments and local variables are present. If any function argument or local variable is not present, a warning is displayed.

To verify the return type, the clang AST function definition is queried for its type. If the YAML representation of the return value of the function is correctly denoted as the same type as the type within the clang AST (assuming the return value is a pointer), then the initial verification is successful. Otherwise, a warning is displayed.

Once verification of the existence of function arguments, local variables, and return type are complete, the `ControlFlowSequence` within the `FunctionDigest` is used to verify the responsibility of function arguments, local variables, and return type. While iterating through the

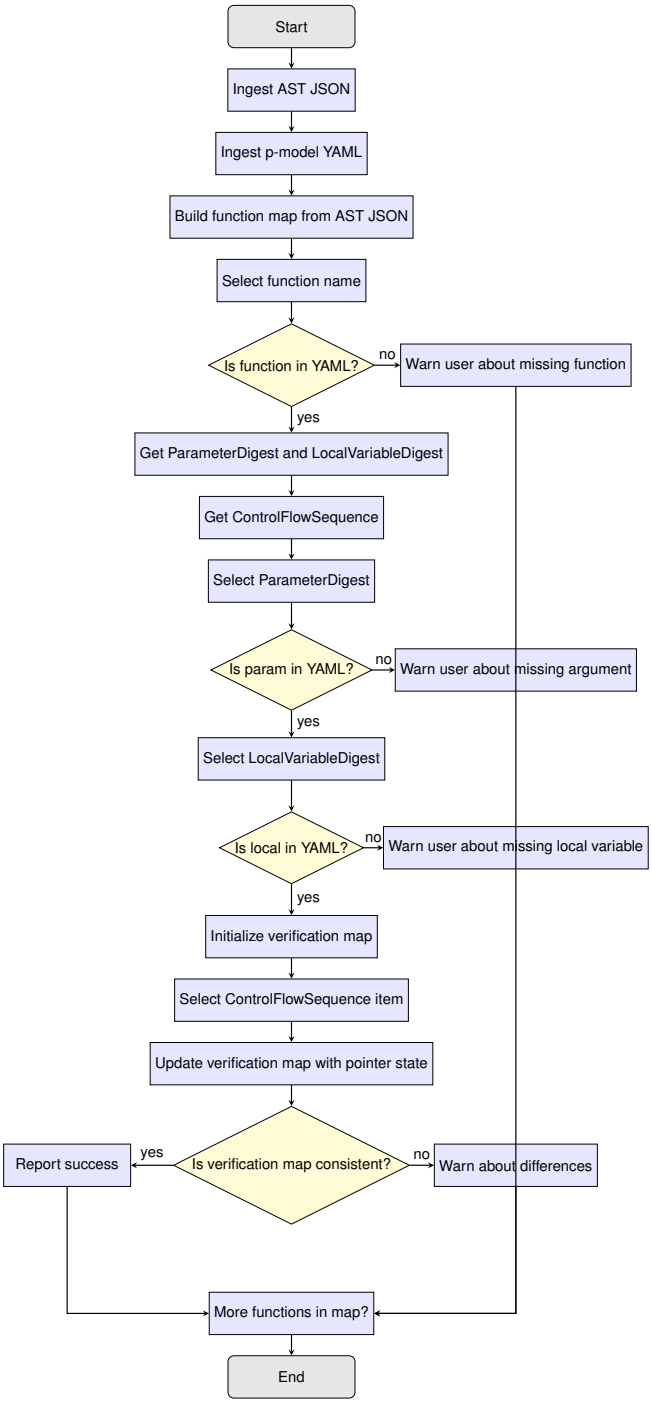

Figure B.1: High-Level Flow of Verifying a p-model

`ControlFlowSequence`, if a control flow item disagrees with the YAML representation of any function arguments, local variables, or return type, then a warning is displayed.

Currently, we have developed verification of the existence of the correct function arguments, local variables, and return type.

References

URLs are valid as of the publication date of this document.

[1] 23andMe. Yamale, June 2025. https://github.com/23andMe/Yamale, original date: 2014-01-27T09:30:25Z.

[2] Robert C. Armstrong, Noah Evans, Geoffrey Compton Hulette, Karla Vanessa Morris, Jon M. Aytac, Philip Alden Johnson-Freyd, et al. Dishwashers of Armageddon: Verifying High Consequence Systems for Nuclear Weapons. Technical Report SAND2019-12574C, Sandia National Laboratories, 2019.

[3] Roberto Bagnara, Abramo Bagnara, and Federico Serafini. C-rusted: The Advantages of Rust, in C, Without the Disadvantages. *arXiv [preprint]*, 2023. DOI: 10.48550/arXiv.2302.05331.

[4] Áron Baráth and Zoltán Porkoláb. Automatic Checking of the Usage of the C++ 11 Move Semantics. *Acta Cybernetica*, 22(1):5–20, 2015.

[5] Kirsten Bradley and Michael Godfrey. A Study on the Effects of Exception Usage in Open-Source C++ Systems. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2019. DOI: 10.1109/SCAM.2019.00010.

[6] Marek Chalupa, Jan Strejček, and Martina Vitovská. Joint Forces for Memory Safety Checking. In *Model Checking Software: 2018 5th International Conference on Signal Processing and Integrated Networks (SPIN 2018)*, pages 115–132. Springer, 2018.

[7] Marek Chalupa, Jan Strejček, and Martina Vitovská. Joint Forces for Memory Safety Checking Revisited. *International Journal on Software Tools for Technology Transfer*, 22(2):115–133, 2020.

[8] Zhe Chen, Chuanqi Tao, Zhiyi Zhang, and Zhibin Yang. Beyond Spatial and Temporal Memory Safety. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 189–190. ACM, 2018.

[9] Zhe Chen, Jun Wu, Qi Zhang, and Jingling Xue. A Dynamic Analysis Tool for Memory Safety Based on Smart Status and Source-Level Instrumentation. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 6–10. IEEE, 2022.

[10] Zhe Chen, Qi Zhang, Jun Wu, Junqi Yan, and Jingling Xue. A Source-Level Instrumentation Framework for the Dynamic Analysis of Memory Safety. *IEEE Transactions on Software Engineering*, 49(4):2107–2127, 2022.

[11] Luis Diogo Couto, Peter W.V. Tran-Jørgensen, René S. Nilsson, and Peter Gorm Larsen. Enabling Continuous Integration in a Formal Methods Setting. *International Journal on Software Tools for Technology Transfer*, 22:667–683, 2020.

[12] cppreference.com. RAII. `https://en.cppreference.com/w/cpp/language/raii`. Accessed May 15, 2025.

[13] DARPA. TRACTOR: Translating all C to Rust. `https://www.darpa.mil/research/programs/translating-all-c-to-rust`. Accessed May 29, 2025.

[14] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 53–60. IEEE Computer Society, 2018.

[15] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, et al. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 608–625. IEEE, 2020.

[16] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Jessica Clarke, Peter Rugg, Brooks Davis, et al. Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 251–268. ACM, 2024.

[17] International Organization for Standardization (ISO) and IEC JTC 1/SC 22. ISO/IEC 9899:2024 – Information Technology – Programming Languages – C. Standard ISO/IEC 9899:2024, October 2024. https://www.iso.org/standard/82075.html.

[18] Kerry Doyle and Twain Taylor. Rust vs. C++: Differences and Use Cases Explained. https://www.techtarget.com/searchapparchitecture/tip/Rust-vs-C-Differences-and-use-cases. Accessed May 15, 2025.

[19] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta Pointers: Buffer Overflow Checks Without the Checks. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018*, pages 22:1–22:14. ACM, 2018.

[20] D. Richard Kuhn, Ramaswamy Chandramouli, and Ricky W. Butler. Cost Effective Use of Formal Methods in Verification and Validation Foundations. In *02 V and V Workshop*. National Institute of Standards and Technology, 2002. https://www.nist.gov/publications/cost-effective-use-formal-methods-verification-and-validation-foundations.

[21] Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. VIP: Verifying Real-World C Idioms with Integer-Pointer Casts. *Proceedings of the ACM on Programming Languages*, 6(POPL):Article 20, 2022. https://dl.acm.org/doi/10.1145/3498681.

[22] Aravind Machiry, John Kastner, Matt McCutchen, Aaron Eline, Kyle Headley, and Michael Hicks. C to Checked C by 3C. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):Article 78, 2022. https://dl.acm.org/doi/10.1145/3527322.

[23] David Manners. CHERI Alliance Launched. https://www.electronicsweekly.com/news/business/cheri-alliance-launched-2024-11/, November 2024.

[24] Jackson R. Mayo, Karla Vanessa Morris Wright, Jon M. Aytac, Andrew Michael Smith, Robert Claude Armstrong, Geoffrey Compton Hulette, and Randall R. Lober. Demonstration of Model-Based Design for Digital Controller Using Formal Methods. Technical Report SAND2024-00442, Sandia National Laboratories, 2024. https://www.osti.gov/biblio/2430067.

[25] MITRE. CWE-401: Missing Release of Memory After Effective Lifetime. https://cwe.mitre.org/data/definitions/401.html, 2006. Accessed May 29, 2025.

[26] MITRE. CWE-415: Double Free. https://cwe.mitre.org/data/definitions/415.html, 2006. Accessed May 29, 2025.

[27] MITRE. CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html, 2006. Accessed May 29, 2025.

[28] MITRE. CWE-476: NULL Pointer Dereference. https://cwe.mitre.org/data/definitions/476.html, 2006. Accessed May 29, 2025.

[29] MITRE. CWE-590: Free of Memory Not on the Heap. https://cwe.mitre.org/data/definitions/590.html, 2006. Accessed May 29, 2025.

[30] MITRE. CWE-908: Use of Uninitialized Resource. https://cwe.mitre.org/data/definitions/908.html, 2012. Accessed May 29, 2025.

[31] Nausheen Mohammed, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. Enabling Memory Safety of C Programs Using LLMs. *arXiv [preprint]*, 2024. DOI: 10.48550/arXiv.2404.01096.

[32] Karla Vanessa Morris, Colin Snook, Thai Son Hoang, Robert C. Armstrong, Geoffrey Compton Hulette, and Michael Butler. Refinement of Reactive Systems. Technical Report SAND2020-7316C, Sandia National Laboratories, 2020.

[33] Karla Vanessa Morris Wright, Thai Son Hoang, Colin Snook, and Michael Butler. Formal Language Semantics for Triggered Enable Statecharts with a Run-to-Completion Scheduling. In *International Colloquium on Theoretical Aspects of Computing*, pages 178–195. Springer, 2023.

[34] Santosh Nagarakatte. Full Spatial and Temporal Memory Safety for C. *IEEE Security & Privacy*, 22(4):30–39, 2024.

[35] Thomas Neele and Anton Wijs. *Model Checking Software*. Springer, 2024.

[36] Samuel D. Pollard, Robert C. Armstrong, John Bender, Geoffrey C. Hulette, Raheel S. Mahmood, Karla Morris, et al. Q: A Sound Verification Framework for Statecharts and Their Implementations. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems*, pages 16–26. ACM, 2022.

[37] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. CN: Verifying Systems C Code with Separation-Logic Refinement Types. *Proceedings of the ACM on Programming Languages*, 7(POPL):Article 1, 2023. https://dl.acm.org/doi/10.1145/3571194.

[38] G. Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

[39] Matthew S. Simpson and Rajeev K. Barua. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. *Software: Practice and Experience*, 43(1):93–128, 2013.

[40] Software Engineering Institute. SEI CERT Coding Standards. https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards, 2024.

[41] David Svoboda and Lutz Wrage. Pointer Ownership Model. In *2014 47th Hawaii International Conference on System Sciences*, pages 5090–5099. IEEE, 2014.

[42] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[43] Maurice H. ter Beek, Rod Chapman, Rance Cleaveland, Hubert Garavel, Rong Gu, Ivo ter Horst, et al. Formal Methods in Industry. *Formal Aspects of Computing*, 37(1):1–38, 2024.

[44] Robert N.M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, et al. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7). Technical Report UCAM-CL-TR-927, University of Cambridge, Computer Laboratory, 2019.

[45] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, et al. CHERIvoke: Characterising Pointer Revocation Using

CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 545–557. ACM, 2019.

[46] Vadim Zaliva, Kayvan Memarian, Brian Campbell, Ricardo Almeida, Nathaniel Filardo, Ian Stark, and Peter Sewell. A CHERI C Memory Model for Verified Temporal Safety. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 112–126. ACM, 2025.

[47] Anna Zamansky, Maria Spichkova, Guillermo Rodriguez-Navas, Peter Herrmann, and Jan Olaf Blech. Towards Classification of Lightweight Formal Methods. *arXiv [preprint]*, 2018. 10.48550/arXiv.1807.01923.

[48] Tong Zhang, Dongyoon Lee, and Changhee Jung. 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*, pages 631–644. ACM, 2019.

[49] Jie Zhou, John Criswell, and Michael Hicks. Fat Pointers for Temporal Memory Safety of C. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):316–347, 2023.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	September 2025	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Design of Enhanced Pointer Ownership Model for C	FA870225DB003

6. AUTHORS
David Svoboda, Lori Flynn, Will Klieber, Ruben Martins, Sasank Venkata Vishnubhatla, Nicholas H. Reimer

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2025-TR-008

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
SEI Administrative Agent AFLCMC/AZS 5 Elgin Street Hanscom AFB, MA 01731-2100	N/A

11. SUPPLEMENTARY NOTES

12A. DISTRIBUTION/AVAILABILITY STATEMENT	12B. DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

This report describes the design for a new temporal memory safety model for C code and an implementation to enforce it. The design improves on CERT's Pointer Ownership Model with enhancements including the use of large language models to complete a per-program model; an improved mechanism to prevent use-after-free errors, inspired by Rust's borrow checker and object lifetimes; improved function argument handling with a new abstraction of diligent or producer arguments; handling structs, unions, or arrays that contain pointers; and correct handling of ambiguity in assignment operations. This report details the research approach and early stage results of designing this model, its extension to C's type system, the tool design methodology, and the design and initial engineering of lightweight specification and validation tools.

14. SUBJECT TERMS	15. NUMBER OF PAGES
model checking, lightweight model checking, static analysis, temporal memory safety, pointer analysis	25

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102