LLM Assistance for Memory Safety

Nausheen Mohammed, Akash Lal, Aseem Rastogi, Rahul Sharma
Microsoft Research, India
Email: mdjnausheen786@gmail.com, {akashl, aseemr, rahsha}@microsoft.com

Subhajit Roy
IIT-Kanpur, India
Email: subhajit@iitk.ac.in

***Abstract*—Memory safety violations in low-level code, written in languages like C, continues to remain one of the major sources of software vulnerabilities. One method of removing such violations by construction is to port C code to a safe C dialect. Such dialects rely on programmer-supplied annotations to guarantee safety with minimal runtime overhead. This porting, however, is a manual process that imposes significant burden on the programmer and, hence, there has been limited adoption of this technique.**

The task of porting not only requires inferring annotations, but may also need refactoring/rewriting of the code to make it amenable to such annotations. In this paper, we use Large Language Models (LLMs) towards addressing both these concerns. We show how to harness LLM capabilities to do complex code reasoning as well as rewriting of large codebases. We also present a novel framework for *whole-program transformations* that leverages lightweight static analysis to break the transformation into smaller steps that can be carried out effectively by an LLM. We implement our ideas in a tool called `MSA` that targets the CheckedC dialect. We evaluate `MSA` on several micro-benchmarks, as well as real-world code ranging up to 20K lines of code. We showcase superior performance compared to a vanilla LLM baseline, as well as demonstrate improvement over a state-of-the-art symbolic (non-LLM) technique.

I. Introduction

Legacy C-programs are pervasive, which makes memory corruption vulnerabilities a major problem for software systems. This problem has attracted a wealth of attention for decades but memory safety violations continue to remain one of the major sources of cyber attacks. From `memorysafety.org`: "Microsoft estimates that 70% of all vulnerabilities in their products over the last decade have been memory safety issues. Google estimated that 90% of Android vulnerabilities in the wild are memory safety issues. An analysis found that more than 80% of the exploited vulnerabilities were memory safety issues."

Researchers have proposed safe dialects of C, such as Checked-C [1], Deputy [2], Cyclone [3], etc. These all use static analysis and lightweight runtime checks to ensure formal memory safety guarantees at low runtime overheads. However, these techniques require source-level annotations. The manual cost of adding these annotations, along with the code rewriting that enable such annotations in the first place, are the main hurdle for adoption of these dialects.

Recently, Large Language Models (LLMs) have shown promise in improving the productivity of software developers [4]. LLMs are highly versatile and accomplish diverse tasks surprisingly well, given the right instructions as prompts. Motivated by their novel capabilities, we present `MSA`, a tool that leverages LLMs to help port C to Checked-C. We are unaware of any prior LLM or Machine Learning based approach for this task. Although recent symbolic analyses have shown promising results [5], our evaluation shows that they miss out on many annotations and do not perform refactoring.

Our main contribution is a novel framework that tightly couples LLMs and symbolic representations. We show a general recipe of breaking whole program transformations into smaller tasks that can fit into LLM prompts, where each task has a code snippet and a symbolic context that contains relevant information about the other parts of the program that are not included in the snippet (Section IV). `MSA` works in multiple phases, where each phase is an instantiation of this recipe for a different task. We present the prompts that we used for each phase (Section V).

LLMs help compensate for shortcomings in symbolic inference techniques by dealing with complex code patterns, as well as performing refactoring where needed to allow for further annotations in the code. For instance, if a procedure signature is modified, then the effects have to be propagated to its caller, possibly transitively. Our evaluation shows that LLMs are able to accomplish this task well when provided a sufficiently detailed prompt. On the other hand, LLMs can hallucinate, thus, using symbolic information where available helps improve overall accuracy. Finally, note that any annotations generated by our tool are checked by the Checked C compiler. Hence, even if the LLM falters and generates an incorrect annotation, memory safety is not compromised—the Checked C compiler will either give a compilation error or instrument the code with a check that will fail at runtime. The higher the accuracy of `MSA`, the less is the amount of time developers have to spend porting to Checked C.

Although this paper focuses on memory safety, we believe our contributions have wider ramifications in formal verification of real-world software. At a high level, any such verification task requires three steps. First step is to rewrite the code so that it is amenable to verification. Second step is to annotate the code with contracts (invariants, type qualifiers, etc.). The third and final step is to check that the code satisfies the contracts, usually with an automated verifier. These steps are interconnected, and any failure in the third step has to be fixed by repeating the first two. While symbolic techniques have been devised towards the second step, for inferring contracts, the first step has generally not received much attention. Given that LLMs have shown great promise in tedious programming tasks, it is a natural research direction to

explore whether they can help with the first two steps, leaving the third untouched in order to avoid compromising soundness. Our work answers this question in the affirmative for the task of memory safety of C.

To summarize, we make the following contributions:

- We present MSA, the first LLM-based assistant for porting C to Checked-C. MSA performs transformations that are out-of-reach of existing (symbolic-only) assistants.
- We present a novel recipe for breaking a whole program transformation into smaller tasks that can fit into LLM prompts.
- We evaluate MSA on real world C-programs, ranging up to 20K lines of code, showing that it can successfully infer 86% of the required annotations correctly.

We plan to open-source the implementation of MSA, along with all the prompt templates that it uses.[1] The rest of this paper is organized as follows. Section II provides a background on Checked C, followed by examples that illustrate the challenges of the porting process from C code. Section III provides background on the state-of-the-art symbolic tool for Checked C inference. Our technical contributions follow next. We provide our generic recipe for whole program transformations using LLMs (Section IV) and then we show how MSA instantiates this recipe to overcomes the challenges in the porting process (Section V). We evaluate MSA (Section VI), discuss threats to validity (Section VII), and survey related work (Section VIII).

II. Challenges in porting C to Checked C

Checked C is a safe dialect of C, inspired from Deputy [2] and Cyclone [3]. It differs from its predecessors in that it allows *checked* and *unchecked* code to coexist. Checked regions guarantee *spatial* memory safety, i.e, any illegal out of bounds memory access is caught and the program is terminated. More precisely, Checked C satisfies a *blame* property where any illegal access can be blamed on the unchecked parts of the code [1].

Checked Pointer Types: Checked C introduces three checked pointer types, namely, `ptr<T>`, `arr<T>` and `nt_arr<T>` in place of the C pointer type `T*`. (These names are abbreviated from their actual names, `_Ptr<T>`, `_Array_ptr<T>` and `_Nt_Array_ptr<T>` for brevity.) The `ptr` type is used for pointers that point to a single object (or null) and are not involved in pointer arithmetic. The compiler inserts a null check at every dereference of a `ptr` type for spatial safety.

The `arr` type is used for pointers that point to an array of values. It is accompanied by a bounds expression that specifies the range of memory that the pointer can access. These declarations appear as `arr<T> p : bounds(lo, hi)` where `lo` and `hi` are expressions that evaluate to the lower and upper bounds of the array, respectively. In addition to the null check, a bounds check of `lo <= (p+i) && (p+i) < hi` is also inserted at every dereference `*(p+i)` for an `arr`. There are other shorthand annotations, such as `count(n)` or `byte_count(n)` that specify the number of elements or bytes that the pointer can access, starting from its current value. These checks are inserted at higher-level passes of the compiler, and may get optimized away by lower-level passes in the compiler if it manages to prove that accesses are within the supplied bounds. This combination of type-assistance and low-level optimizations makes the Checked C approach appealing compared to other safe C approaches; portions of the FreeBSD kernel that were ported to Checked C reported essentially no runtime overhead [6].

The `nt_arr` type is used for pointers that point to null-terminated arrays, mostly strings. An annotation of `count(n)` implies that the array has at least $n+1$ elements, the last being the null value. An interesting feature of `nt_arr` is that its bounds can be widened until a null character is found. Hence, `while(*p != 0) p++;` is a valid way to access `nt_arr<T> p:count(0)`. Each time `*p` does not equal the null character, its bound can be widened by 1.

Challenges: Next, we explain some of the challenges in porting C to Checked C through examples. In the changelogs below, the C-code in red needs to be replaced with code in green for a successful port. The examples in this section are derived from real world C programs used in our evaluation, and they are beyond the capabilities of existing inference tools (Section III).

The first challenge is in handling nested pointers. Given a nested pointer, e.g., `long** pt`, it is not possible in Checked C to separately annotate the buffers `pt[0]`, `pt[1]`, etc., with their sizes. The idiomatic way to handle this situation is to replace the nested pointer with an array of structs, where the struct has a buffer with its associated length. Correspondingly, every access to `pt` throughout the program must be modified to respect this new interface. A real-world example follows:

```
void AllocAssign(void) {
  ulong net; ulong n = channelNets+1;
  costMatrix =
-   (long**) malloc(n * sizeof(long *));
+   (arr<struct arr_of_long>)
+   malloc(n * sizeof(struct arr_of_long));
  for (net = 1; net <= channelNets; net++) {
- costMatrix[net] =
+ costMatrix[net].ptr =
    malloc((channelTracks+2) * sizeof(long));
+ costMatrix[net].len = channelTracks+2;
  }
}
```

Here, `costMatrix` has been converted from a nested pointer to an array of structs, where each struct has a buffer `ptr` and its associated length `len`. Whenever the buffers in `costMatrix` are allocated or reassigned, the newly introduced size field `len` must be updated accordingly.

Next, we present an example where annotating a buffer with its bounds requires involved arithmetic reasoning within a loop. Consider the following code where the loop runs `longs` number of times with 4 bytes in the buffer `buf` accessed per loop iteration:

[1] See https://aka.ms/checkedc-annotation-inference.

```
static void byteReverse(
- unsigned char* buf,
+ arr<unsigned char> buf: count(longs * 4),
unsigned longs) {
  uint32_t t;
  do {
    t = (uint32_t)
    ((unsigned) buf[3] << 8 | buf[2]) << 16 |
    ((unsigned) buf[1] << 8 | buf[0]);
    *(uint32_t *) buf = t; buf += 4;
  } while (--longs);
}
```

By analyzing this loop, we can conclude that `buf` has a size of `count(longs * 4)`. Existing tools [5] struggle to infer bounds that have expressions with arithmetic. While the above loop had deterministic behavior, the following is more complicated.

```
struct bin_to_ascii_ret
vsf_ascii_bin_to_ascii(
- const char* p_in,
+ arr<const char> p_in: count(in_len),
- char* p_out,
+ arr<char> p_out: count(in_len * 2),
 unsigned int in_len, int prev_cr) {
 ...
 while (indexx < in_len) {
   char the_char = p_in[indexx];
   if (the_char == '\n' && last_char != '\r'){
     *p_out++ = '\r';
     written++;
   }
   *p_out++ = the_char;
   indexx++;
   ...
```

Here, the buffer `p_in` must be annotated with its length `in_len` and, for memory safety, `p_out` must have a size of `(in_len*2)` to handle the worst case loop behavior.

Often the annotation process is not local and requires refactoring several functions. Consider the following code:

```
static int countint (lua_Integer key,
- unsigned int* nums
+ arr<unsigned int> nums: count(count_nums),
+ int count_nums
) {
    unsigned int k = arrayindex(key);
    if (k != 0) {
      nums[luaO_ceillog2(k)]++;
      ...
```

Just looking at the original C code of `countint`, it is impossible to annotate the buffer `nums` with its size. The programmer likely has some custom invariant in mind to keep the indexing within bounds. It is not possible to explain custom invariants to the Checked C compiler. Thus, the idiomatic way to enforce safety with Checked C is to introduce a new argument `count_nums` that stores the array size and use it to annotate `nums`, then require callers to pass the appropriate size. Let's consider a caller of `countint` next.

```
static int numusehash (
- const Table* t,
+ ptr<Table> t,
- unsigned int* nums,
+ arr<unsigned int> nums: count(count_nums),
+ int count_nums,
  unsigned int* pna) {
  ...
- ause += countint(keyival(n),nums)
+ ause += countint(keyival(n),nums,count_nums)
  ...
```

Here, the call to `countint` must be modified to match its new signature. Furthermore, `numusehash` must annotate `nums` with its size in its signature. Accordingly, the signature also needs an extra argument `count_nums`. Next, the callers of `numusehash` also need to be updated to supply this extra argument. We show one such caller next:

```
static void rehash(lua_State* L,
- Table* t,  const TValue* ek
+ ptr<Table> t, ptr<const TValue> ek
) {
  unsigned int nums[MAXABITS + 1];
  ...
- total += numusehash(t,nums,&na);
+ total += numusehash(t,nums,MAXABITS+1,&na);
  ...
- na += countint(ivalue(ek),nums);
+ na += countint(ivalue(ek),nums,MAXABITS+1);
  ...
}
```

In the above code, the new parameter `count_nums` is set to its correct value `MAXABITS+1` in the calls to `countint` and `numusehash`. We are unaware of any prior tool that performs such a whole program refactoring of C-code automatically.

III. Background on Symbolic Inference with 3C

3C [5] is a static-analysis-based tool to help developers port C code to Checked C. It consists of two components, a type inference algorithm called typ3c and a bounds inference algorithm called boun3c, that are executed one after the other. We briefly discuss these algorithms, their strengths and limitations since `MSA` uses 3C.

typ3c uses type qualifier inference [7] to convert legacy pointers to checked pointer types. Pointers that are used unsafely, for instance, in unsafe casts (e.g., casting an `int` to `int*`), are not converted to checked pointers. This is because the Checked C compiler cannot provide safety guarantees for such pointers. typ3c then classifies checked pointers into one of `ptr`, `arr`, and `nt_arr` depending on their use.

Once the checked pointers have been identified, boun3c infers bounds for `arr` and `nt_arr` pointers. It works by constructing a dataflow graph that tracks the flow of arrays along with their bounds, starting from their allocation site where the bounds information is available. boun3c is designed to be sound (i.e., inferred annotations are correct) but not complete (not all possible annotation are inferred). In fact, as our evaluation will show, boun3c misses several annotations.

One limitation of boun3c is that it can only infer bounds that involve a single variable or a constant. Thus, bounds that involve expressions with arithmetic will be missed for sure. Further, boun3c is largely limited to inferring `count`-style

annotations, not `bounds` annotations that have both a lower and upper bound. boun3c offers some heuristics that attempt to come up with more annotations, but these annotations can be unsound. Furthermore, in some cases, no good bound exists without rewriting the program, which boun3c is not prepared to do.

Note that 3C, as a combination of typ3c followed by boun3c, does not attempt to fully port from C to Checked C. The resulting code, for instance, is not guaranteed to pass the Checked C compiler. This is because several bounds annotations might still be missing, as well as some dynamic casts might be necessary to pass the type checker of Checked C. 3C is, thus, an assistance in the porting process. We carry on with this design philosophy in this paper, striving to provide even more assistance without guaranteeing a complete port to Checked C.

IV. Whole-program transformation with LLMs

As illustrated in Section II, the task of porting to Checked C requires making several changes throughout the program. Even with the increasing prompt sizes, it is still unreasonable to expect entire code to fit inside a single prompt. Furthermore, we found that even when we can fit larger parts of a program in a single prompt, the accuracy of an LLM is lower when asked to make several changes, compared to doing fewer changes to a smaller piece of code (see Section VI).

The challenge then is to break a whole-program transformation into multiple smaller tasks. With LLMs, each inference query is treated independently of ones made previously, therefore, any query on some part of the code must provide enough context about the rest of the code in order to carry out the task effectively. We address these challenges through static analysis.

A. Dependency Graph Generation

We use a lightweight static analysis that goes across all input source files and constructs a data structure that we call as a *dependency graph*. Nodes of this graph are all the top-level declarations in the program. These can either be procedures (both its signature and its body), type declarations (`struct`, `union`, `enum`), global variable declarations or macro definitions. There is a directed edge from node n_1 to n_2 if n_2 is directly *used* by n_1, as follows:

- **Procedures.** If n_1 is a procedure, then we place out-going edges from n_1 to all procedures that are directly called by it, as well as all types, globals and macros that appear somewhere inside n_1. We only consider direct calls. For indirect calls through a function pointer, there will be an edge to the type declaration of the pointer's type, not to potential targets of the pointer.
- **Types.** If n_1 is a type definition, then we place out-going edges to all types and macros that appear in the definition.
- **Globals.** If n_1 is the declaration of a global variable, then we place an out-going edge to the type of the variable or any macro that the declaration might use.

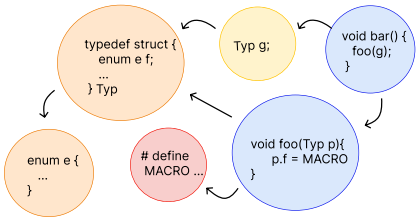

Fig. 1: Example of a dependency graph

- **Macros.** There are no out-going edges from macros. Transformation of macros is currently outside the scope of our analysis. We didn't find a need for it in our experiments.

The requirements of this analysis are purposefully kept simple for ease of implementation. We use `clang` to parse and construct ASTs of all input source files. We then perform linking at the AST level to resolve procedure calls and dump the dependency graph; see Figure 1 for an example.

As part of this analysis, we also record additional information for each node that can be obtained easily from its AST representation. For instance, for a procedure, we keep track of its signature, argument list, type of each argument, type of each local variable, etc.

B. Generic Whole-Program Transformation

Porting to Checked C requires not just adding annotations, but also supporting edits that allows for the presence of an annotation in the first place. Inspired from previous experience of porting C to Checked C (like in [1], [8], [5]), we define three different programs transformations, which are applied sequentially in order. Each of these transformation follow a common structure, shown in Algorithm 1.

Algorithm 1 Whole-Program Transformation with LLMs

```
1:  procedure ProgramTransformation(𝒟, 𝒯)
2:      for d ∈ Nodes(𝒟) do
3:          refactored[d] := False, oldcode[d] := d.code
4:      end for
5:      for d ∈ BottomUpOrdering(𝒟) do
6:          prompt ← PromptTemplate( )
7:          prompt.task ← TaskDescription(𝒯)
8:          prompt.example ← TaskExample(𝒯)
9:          prompt.prelude ← d.succ
10:         prompt.code ← d.code
11:         prompt.refactor_history ← {(oldcode[u],
    u.code): u ∈ d.succ, refactored[u] }
12:         prompt.elements ← TaskElements(d, 𝒯)
13:         response ← LLM(prompt)
14:         d.code ← ApplyPatch(response, d.code)
15:         refactored[d] ← SignatureChanged(response)
16:     end for
17: end procedure
```

Algorithm 1 takes the dependency graph ($\mathcal{D}$) of the input program, as well as a description of task-specific information

```
% (1) CheckedC Preamble
Checked C has three checked pointer types that support
following annotations:
...

% (2) Task definition
{{Task definition}}

% (3) Propagate changes
Similar changes have been made in other parts of the
code. Given the refactor history, update the current
code accordingly.

% (4) Output format
Each change must be outputted as a block with original
lines and refactored lines in the below format. Output
a series of such blocks, one for every change.
...

% (5) Example
Consider this example input and output as a reference.
{{Task example}}

% (6) Code
Here is relevant context for the given code
{{prelude}}

This is the code that must be transformed
{{code}}

This is a history of the previous changes
{{refactor_history}}

Perform the given task on these parts of the code:
{{task_specific_code_elements}}
```

Fig. 2: Prompt template for Whole-Program Transformation

($\mathcal{T}$), and outputs a new program that is the result of applying $\mathcal{T}$ to the input program. The algorithm goes over the input program one declaration at a time, and instructs an LLM to preform a rewriting according to $\mathcal{T}$.

The template of the LLM prompt is shown in Figure 2. It consists of various sections. The first section is a preamble on Checked C that defines its various annotations (e.g., `arr`), their meaning (e.g., it is a pointer to an array) as well as the syntax rules to follow (e.g., bounds annotation appears after a colon).

The next section of the prompt (2) carries a description of the task $\mathcal{T}$ (e.g., "infer bounds of arrays"). Next (3) is an instruction telling the model that prior edits have been made to the program, and it must edit the given snippet to respect those refactorings. For instance, when a callee method signature is modified, the call to that method must be modified as well.

Section 4 of the prompt defines an output format that the model should follow. Instead of asking the model to produce the entire modified code, prior work has found it useful to instead ask for a "diff" or a patch that can be applied to the original code. We simply follow the prompt used in prior work [9] to obtain a patch from the LLM. Any other format or formatting instructions can be used as well.

Section 5 of the prompt is an example of the task. For each task, we only include one or two fixed hard-coded examples. Section 6 of the prompt includes the relevant code snippets from the input program. This includes context for the current code (called "prelude"), the code that must be transformed

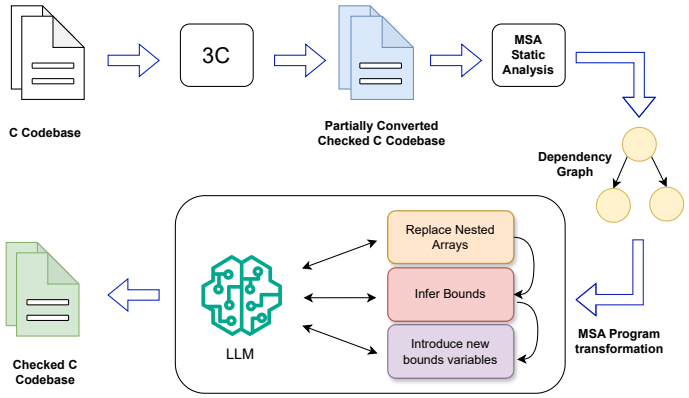

Fig. 3: Workflow of porting C code to Checked C with `MSA`.

(code) as well as previous code refactorings (refactor_history). Finally, the prompt also includes some code-specific elements. For instance, for the task of inferring bounds of arrays, we explicitly list the variable names with array types in order to help focus the attention of the model on those variables.

Getting back to Algorithm 1, it starts (lines 2 to 4) by keeping track of the original code (`oldcode`) as well as remembering what parts of the code have been refactored (`refactored`), initially none. It then makes one pass over all declarations in the code. These declarations are picked in a bottom-up order, using a reverse topological sorting of the dependency graph. In general, the dependency graph can have cycles because of recursive types or recursive procedures; we break these cycles arbitrarily in order to limit the transformation to a single pass over the program text.

For each declaration d, the prompt in instantiated with task-specific as well as code-specific details. Code prelude is computed as all immediate successors of d in $\mathcal{D}$, i.e., all code elements that are referenced directly in d. For brevity, when including a procedure in the prelude, we only include its signature and not its body. The refactor history includes all changes made to these successors of d so far, if any.

When the LLM is prompted, it returns a patch (possibly empty) that should be applied to d. We apply this patch to update d. Finally, we set `refactored`[d] to true if the patch was non-empty, i.e, d was updated. When d is a procedure, we set `refactored`[d] to true only when the signature of d was changed by the patch.

V. `MSA` Design and Implementation

The design of `MSA` is illustrated in Figure 3. The tool is a combination of symbolic as well as LLM-based components. As a design principle, we rely on symbolic components whenever they exist or are easy to implement. Tasks that are either complex to do symbolically (e.g. knowing if a code pattern respects null-termination of a given array) or require an LLM's flexibility (e.g., refactoring to accommodate a new struct field) are left to LLM-based components.

`MSA` feeds input C code to the 3C tool to produce partially-converted CheckedC code. 3C is quite fast (especially com-

```
You are given a list of Checked C declarations and a
partially converted Checked C code snippet. Array of
arr<T> is not supported in Checked C. Your task
is to replace them with an array of struct having a
pointer field 'ptr' and a bounds field 'len'. You will
also have to replace the uses of the nested array with
the uses of the struct 'ptr' field instead. Make sure
to update the 'len' field whenever the 'ptr' field is
updated.
```

Fig. 4: Task description for replacing nested arrays with structs

From:

```
int foo(arr<arr<int>> a, int i) {
  return a[i][i];
}
```

To:

```
// New struct
typedef struct arr_of_int {
  arr<int> ptr : count(len);
  int len;
} arr_of_int;
// type of a changes
int foo(arr<struct arr_of_int> a, int i) {
  // nested pointer access via the ptr field
  return a[i].ptr[i];
}
```

Fig. 5: An example of the transformation ("**From**" to "**To**") for nested arrays that is provided to the LLM. The code comments are included as well.

pared to LLM inference times), thus it serves us well to use it as a source of cheap and sound annotations. We turn off the heuristics in boun3c, which we observed to produce incorrect annotations. We leave heuristics to the LLM-based components of `MSA`. The role of `MSA`, thus, is to annotate the unannotated checked pointers in the program produced by 3C.

`MSA` takes the output of 3C, constructs its dependency graph, and uses it in subsequent LLM-based program transformations. `MSA` is parametric on the choice of the LLM, although we only evaluate with GPT4. In order to account for the randomness in the LLM's response, `MSA` asks for multiple completions (i.e., responses) for each LLM query. The default setting is 10, although it can be changed by the user depending on their time budget. Multiple completions produce multiple code patches; `MSA` takes a majority vote among these patches.

We now describe the three program transformation tasks.

A. Replacing Nested Arrays with Structs

As mentioned in Section II, the type `arr<arr<T>>` is not allowed in Checked C syntax. The recommended way is to replace it with `arr<struct arr_of_T>`, where `struct arr_of_T` is a new structure with two fields, one of type `arr<T>` for storing the inner array, and the other of type `int` for storing the bound of this array. The rest of the program should change to use this structure. For instance, the bounds field must be updated when the corresponding array field is set or updated.

```
Determine and assign 'count(..)' or 'bounds(.., ..)'
expressions for each att and nt_arr in the
given function. To find valid bounds for a pointer p,
examine all uses of p and set bounds that encompass every
access. Alternatively, adopt the bounds from the pointer
from which p was assigned.

You will be provided a list of pointer variable names
along with their declaration line number. You must
choose one of the following rules for each of them.

[A0] Infer a valid bounds expression:
    Provide a 'count(..)' or 'bounds(..,..)' expression
    at the line of declaration. Choose this only when
    you are completely sure that the bounds are valid.

[A1] Say unknown:
    When there is not enough information to infer bounds
    for a pointer, it is okay to leave the annotated line
    same as the original line. Follow this by explaining
    why enough information is not available. This can be
    chosen when there is not a clear upper bound to all
    accesses through the pointer or the pointer depends on
    other pointers whose bounds are not known.

[A2] Change an arr to nt_arr:
    If you cannot infer the bounds to arr p but you do
    know that p is terminated with a null character from
    its use, you can change its type to nt_arr.
    Make sure to also change the pointers that p was
    derived from to nt_arr in such a case. This can
    also be due to a callee now taking nt_arr
    instead of arr due to an earlier refactor.

[A3] Add a parameter for bounds:
    If you cannot infer a reasonable bound for a pointer
    parameter, add a new parameter to store its bounds and
    use that in the bounds expression. Going ahead, all
    calls of this function will have to be passed this
    extra bounds argument.
```

Fig. 6: Task description for inferring bounds

Figure 4 shows the description for this task that is used in conjunction with the template shown in Figure 2 to carry out the program transformation. `MSA` first adds the declaration of `struct arr_of_T`, once for each type `T` such that the type `arr<arr<T>>` appears in the input program. This step is done symbolically, at the AST level. `MSA` then uses Algorithm 1 to carry out the required transformations in the program to use this new type. Figure 5 shows the example that is included in the prompt. The task-specific elements that are included in the prompt are the names of variables with refactored types (e.g., `a` for the example in Figure 5).

B. Inferring Bounds Annotations

The second transformation does the actual inference task of adding bounds annotations for `arr` and `nt_arr` pointers. The LLM is asked to annotate array pointers based on their usage in the code snippet that is presented to it. Because this pass traverses the code in a bottom-up fashion, when the LLM is presented the code of a certain procedure, it will also get presented with its context. This context will include callee signatures, which have already been annotated because they came before in the dependency-graph order. This allows transitivity of the inferred annotations.

The task description for this transformation is the most detailed among the three passes and is shown in Figure 6. It

From:
```
struct x { int f; int g; }
int foo(arr<struct x> a, int i) {
  int j = a[i].f;
  arr<struct x> p = a;
  return a[j].f;
}
```
To:
```
// [A3] As j is read from the heap, the access
// a[j] could be anything. Moreover, j is not
// in scope at line 1. Since 'a' is a pointer
// parameter, add a bounds parameter instead
// of saying 'unknown'.
int foo(arr<struct x> a : count(count_for_a),
  int count_for_a, int i) {
  int j = a[i].f;
  // [A0] As p is assigned a, the bounds for a
  // are valid for p too.
  arr<struct x> p : count(count_for_a) = a;
  return a[j].f;
}
```

From:
```
void foo() {
  char a[10]; nt_arr<char> p = a;
}
```
To:
```
void foo() {
  char a[10];
  // [A0] When an array is converted to nt_arr
  // the count is the size of the array - 1.
  nt_arr<char> p : count(9) = a;
}
```

Fig. 7: Examples of inferring bounds for `arr` and `nt_arr`

lists down four different rules ($A0$—$A3$). The first rule ($A0$) is for inferring bounds "when the model is sure of it" and the second rule ($A1$) provides an escape hatch for the same. The idea behind $A1$ is to reduce hallucinations when the model is not confident of inferring bounds; that is, the model should choose to leave things unannotated rather than add an incorrect one. The third rule ($A2$) rectifies inaccuracies in typ3c where it fails to identify that certain arrays are null-terminated. In our experience, we found that this happens because typ3c is sometimes unable to identify that a comparison to the null character is being used to break out of a loop through the array. In other cases, it lacks an understanding of standard string operations (from `stdlib`), which also establish that a given array is (intended to be) null-terminated. An LLM can compensate for these limitations by promoting `arr` to `nt_arr`, helping gain information about the length of the array. Finally, the last rule ($A3$) instructs the model that if the bound of a parameter is not obvious from its use in the code, then it should add a new parameter to the procedure, and pass the obligation to the callers (which are yet to undergo transformation) to pass the appropriate bounds in the newly added parameter. Note that the prompt only says what annotations it wants, it does not say how to obtain them. The complexity of actually doing program analysis is completely left to the model.

```
You are given a Checked C code snippet, with a history of
refactors. The refactors introduce a new variable to
store the bounds of a pointer variable, which can be a
struct field or a global variable. Update the newly
introduced bounds variable with the correct bounds
whenever its corresponding pointer variable is assigned a
new value. Make the update in the same statement as the
assignment.
```

Fig. 8: Task description for adding new bounds variables

Figure 7 shows two examples that are provided to the model. The first example shows applications of rules $A0$ and $A3$ for `arr`, and the latter example shows an application of rule $A0$ for `nt_arr`, where it makes a note of the off-by-one computation for bounds of null-terminated arrays. In terms of program elements that are provided in the prompt of Figure 2, we give variable names of all `arr` and `nt_arr` typed variables that are currently unannotated.

`MSA` follows Algorithm 1 with the above task description to make changes to the program, with a few minor changes. It restricts the bottom-up traversal to only the procedures in the program, not the other top-level declarations. That is, it goes bottom-up on the program call graph. As mentioned in Section IV, we break cycles in the call graph arbitrarily. This implies that the inference accuracy can suffer for mutually recursive procedures. It is an interesting future work to investigate the use of a fix-point iteration over the mutually-recursive cycle to improve accuracy. Further, when looking at a particular procedure p, the model is also allowed to add annotations on any globals or struct fields that p uses, which are anyway present in the prelude of p. Finally, globals and struct fields can be used in multiple procedures, each of which are only considered one-by-one. Thus, it is possible that different procedures produce conflicting annotations for globals or struct fields. In this case, `MSA` detects this conflict in a post-processing step and drops the corresponding annotation. These missing annotations are left for the third transformation that follows next.

C. Annotating Globals and Struct Fields

Annotations for global variables as well as struct fields have a global scope. That is, these annotations are expected to hold throughout the lifetime of the program. Consequently, it is possible that the previous pass, which only consider one procedure at a time, fails to infer a consistent bound for them. In these cases, the third pass takes over.

`MSA` creates a new variable for storing the bounds of the respectively element. In particular, it creates a new int-valued global variable called `count_for_g` for each global variable `g` of type `arr` that is unannotated so far. It adds the annotation `count(count_for_g)` to the declaration of `g`. `MSA` also creates a new int-valued field called `count_for_f` for field `f` (in the same struct) of type `arr` that is unannotated so far. It adds the annotation `count(count_for_f)` to the field `f`. These changes are made symbolically by `MSA` and

From:

```
void foo(arr<struct x> a, int i) {
  a[i].p = malloc(sizeof(int) * 10);
}
```

To:

```
struct x {
  int count_for_p;
  arr<int> p: count(count_for_p);
}

void foo(arr<struct x> a, int i){
  a[i].p = malloc(sizeof(int) * 10),
  a[i].count_for_p = 10;
}
```

Fig. 9: Example of transformation for globals and struct fields

added to the refactor history. MSA then runs Algorithm 1 with the task description shown in Figure 8. This transformation instructs the model to update the bounds variable each time the corresponding array is assigned. Figure 9 shows the example that is provided in the prompt. In terms of program elements, MSA provides the names of the global variables and struct fields for whom the additional variables have been introduced.

VI. Evaluation

In this section we evaluate MSA empirically. We use gpt-4-32k from Azure OpenAI Service as the LLM in these experiments. We divide our experiments into two categories: (a) benchmarking effectiveness of the different components of our algorithm and the prompt template, and (b) evaluating the effectiveness of MSA in inferring Checked C annotations in real-world codebases. We also present our experience on porting vsftpd, one of our real-world benchmarks, end-to-end.

A. Benchmarking the MSA algorithm

We vary different components of our algorithm and compare the results to show their relative effectiveness. We consider: (a) the background about Checked C and specific inference instructions in the prompt template, and (b) modular inference with dependency-order traversal of the codebase.

For this experiment, we use a subset of the Olden [10] and Ptrdist [11] benchmarks, with code sizes that allow the entire program to fit in a single prompt. For large codebases, modular inference is a necessity, however, we explore if modularity helps even for small programs that may fit in a single prompt. We define a trivial algorithm that uses a prompt, consisting of some instructions followed by a full C program, and queries the LLM to obtain its output. Within this algorithm, we vary the instructions part of the prompt with the following versions:

- **V0**: no background on Checked C or specific inference instructions to the LLM,
- **V1**: background on Checked C and inference instructions, as explained in Section IV.

Version **V0**'s performance was extremely poor, with the LLM often bailing out either by citing the problem as too hard

	Total	boun3c	V1 (non-modular)		MSA	
			Inf	Correct	Inf	Correct
mst	11	8 (72%)	5	3 (27%)	11	11 (100%)
power	7	7 (100%)	1	1 (14%)	7	7 (100%)
em3d	22	13 (59%)	13	11 (50%)	22	17 (77%)
anagram	13	2 (15%)	7	4 (30%)	13	11 (84%)

TABLE I: Comparison with boun3c and a non-modular version of MSA on Olden (mst, power, and em3d) and Ptrdist (anagram) benchmarks. The numbers in parentheses indicate the percentage of total annotations inferred correctly.

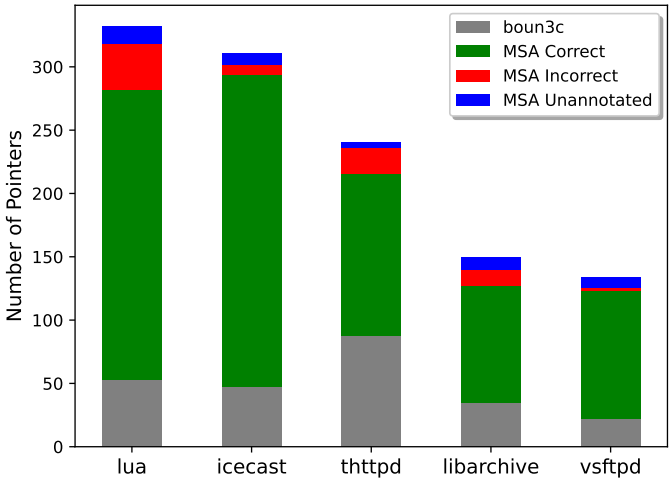

Fig. 10: MSA performance on real-world codebases.

or by saying there are no pointers to be annotated. Results for **V1** and MSA are shown in Table I (it also shows numbers for boun3c). For each benchmark, the table shows the total number of bounds annotations required, the number of annotations inferred by the algorithm (Inf), and the number of correct annotations amongst them (Correct). For determining which of the inferred annotations are correct, we manually prepare annotated versions of these benchmarks that compile with the Checked C compiler and pass the runtime tests. Then, an inferred annotation is defined to be correct if it matches the manual annotation in these ground-truth versions.

As the table shows, while MSA correctly infers 86% of the annotations on-average, the non-modular version **V1** infers only 35%. There are 5 cases across these benchmarks where refactoring instructions—absent from **V1** but present in MSA—play a role. Otherwise, the only difference is modular analysis.

Conclusion. Without the Checked C background and inference instructions in the prompt, LLMs cannot infer annotations even for small programs. With instructions, modular analysis is more effective than fitting the entire (small) program in a prompt. Thus, it is better to focus LLM on one procedure at a time, providing it the dependencies in the context.

B. Inferring annotations in real-world codebases

Table II shows the details of real-world benchmarks that we picked from 3C experiments [5], where 3C could not infer a reasonable number of annotations. These benchmarks cover a

	LOC	RB	b3c	b3ch	Remaining
lua	19.4K	332	53	31	279 (126 + 153)
icecast	18.2K	311	47	5	264 (26 + 238)
thttpd	7.6K	240	88	7	152 (31 + 121)
vsftpd	14.7K	134	22	15	112 (26 + 86)
libarchive	146.8K	150	35	3	115 (63 + 52)

TABLE II: Benchmark details. RB is the number of required bound annotations. b3c is the number of annotations inferred by boun3c, while b3ch is the number of additional annotations inferred by boun3c heuristics. Remaining (divided into `arr` and `nt_arr`) is the difference of RB and b3c.

	Functions	Globals, structs	Queries	Input, Output size
lua	10	7	246	2627, 1895
icecast	45	2	192	2998, 1988
thttpd	82	6	104	2861, 3441
vsftpd	6	0	111	2322, 1600
libarchive	58	89	805	3360, 1847

TABLE III: Number of refactorings applied by `MSA`, total number of queries made to the LLM and average number of input and output tokens per query

variety of different domains (language interpreter, http server, ftp server, media server, compression library, etc.) and come from a list recommended by the Checked C community [12]. For each benchmark, the table shows the size of the benchmark (in terms of lines of code), total number of pointers that required an annotation (RB), the number of pointers annotated by boun3c (b3c), additional pointers that boun3c heuristics infer (b3ch), and the number of remaining unannotated pointers (RB - b3c). We discount the boun3c heuristics from rest of our experiments; first, the number of additional annotations that they infer over boun3c is not substantial, and second, on manual inspection, nearly a third of them seemed incorrect to us. Hence, we consider Remaining as the target for `MSA`. For `libarchive`, we ran `MSA` on the whole codebase, but analyzed only a subset of the annotations for correctness (150 unannotated pointers), due to time constraints.

Figure 10 plots the performance of `MSA` on these benchmarks. We categorize the inferred annotations into Correct and Incorrect based on a manual inspection of the inferred annotations. In total, `MSA` infers 877 annotations across all benchmarks, out of which 797 annotations are correct and 80 are incorrect. Thus, `MSA` correctly infers 86% of the annotations remaining after boun3c (797/922). Of the 797 correctly inferred annotations, 212 are array pointer bound annotations (78% of remaining array pointer bounds) while 585 are NT array bounds annotations (90% of remaining NT array bounds); of these 585 NT array annotations, 532 (91%) are `count(0)` annotations whereas only 9% (26/272) of the array pointer bounds annotations missed by boun3c are `count(0)`.

Below is a code snippet from `icecast` that shows a couple of annotations that `MSA` infers but boun3c could not:

```
// MD5Update(...,
//   arr<char> buf:count(n), int n);
//
// util_bin_to_hex(..., int n)
//   : nt_arr<char> count (n * 2);
nt_arr<char> get_hash
  (const char *data: nt_arr<char>, int len) {
  ...
  MD5Update(&context, data, len); ...
  return (util_bin_to_hex(digest, 16));
}
```

`MSA` constructs the context for `get_hash`, shown as comments, and from this information, it is able to infer that `data` has count `len` and the returned value has count `32`. boun3c is unable to infer the return type annotation because it is not able to infer the `n*2` annotation for `util_bin_to_hex` (it doesn't support arithmetic expressions in annotations). It is also unable to infer annotations for `data` as one of the call sites for `get_hash` is `get_hash(p, strlen(p))`, and boun3c is unable to correlate the two parameters of the call.

Following is an example from `thttpd` where `MSA` successfully changes an `arr` annotation from typ3c to `nt_arr`.

```
static void defang(arr<char> str, ...) {
  arr<char> cp1 = (void *) 0;
  for(cp1 = str; *cp1 != '\0'; cp1++) {
    // access *cp1
  }
```

Here, typ3c marks the `str` argument and `cp1` as `arr<char>`. However, analyzing the access patterns (loop iterating until the null character), `MSA` infers an `nt_arr<char>` type with `count(0)` annotation for both of them. There were also multiple instances when `MSA` refactors the code to allow additional annotations. For example:

```
void luaL_setfuncs (arr<const luaL_Reg> l) {
  for (; l->name != NULL; l++) { ... }
}
```

In this function, from `lua`, the array pointer `l` is incremented until its `name` field is `NULL`. Since there is no reasonable annotation for it in the code as written, `MSA` adds a bounds argument `l_count` to the function and the annotation `count(l_count)` to `l`.

Out of the 877 annotations that `MSA` infers, 80 (9%) are incorrect. Most of these are subtle cases where pointers are accessed with unusual patterns. E.g., in the following snippet from `lua`, the pointer is accessed with a negative index:

```
copy2buff(StkId top:arr<StackValue>, int n) {
  do {
    // use *(top - n)
  } while (--n > 0);
}
```

`MSA` infers `count(n)` for the `top` pointer, whereas the correct annotation would be `bound(top - n, top)`. Improving performance on such code patterns is future work.

The annotations that `MSA` does not infer (45/922) are mostly due to the $A1$ rule of Figure 6. In very few cases, the LLM produced annotations that referred to variables that are not in scope; these are dropped automatically by `MSA` as a post-processing step.

Table III shows the number of function call and globals and struct refactorings that MSA applies per benchmark. The column Functions includes both the number of function parameters added as well as modifications to respective function calls. Similarly, the column for globals and structs include both the number of global variables and struct fields added as well as their corresponding assignments. For real-world codebases, the nested pointers to struct transformation did not appear (while one case of this appears in Table I).

Table III also shows the number of LLM queries required per benchmark, and the average number of input and output tokens (including all completions) per query.

Conclusion. When porting real-world codebases to Checked C, 3C leaves a substantial number of pointers unannotated. MSA is able to infer majority of these pointers (86% in our experiments). We observe that MSA is able to infer annotations that require sophisticated code reasoning. We also observe that porting is not just about inferring annotations; code edits, function refactorings, globals and struct refactoring are also commonplace. Their support in MSA is important for real-world codebases.

C. vsftpd: End-to-end case study

We present a case study on end-to-end porting of vsftpd. In this exercise, we take the output of MSA and make further edits so that it successfully compiles with Checked C. We started by reverting the incorrect annotations and adding the missing annotations in the MSA output. This includes the red and blue regions in the vsftpd bar in Figure 10; we corrected 3 incorrect annotations requiring 6 edits and added 8 missing annotations requiring 22 edits. The remaining work involved passing the Checked C compiler. For that, we had to change a further 148 lines in the code. These changes are the known caveats in making a codebase compile with Checked C [13]. For example, we had to add dynamic bound casts (that are checked at runtime) and assume casts, when the Checked C compiler could not reason about the bounds due to, e.g., lack of flow-sensitivity. An example is as follows:

```
// p : nt_arr<char> count(0)
if (p[0] != '-')
    {...}
else
    { // access p[1] }
```

In the else branch, accessing p[1] is safe because we know that p is an nt_arr and that p[0] is not null. However, the Checked C compiler is unable to reason about this, so we added an assume bound cast to bypass the checker. In a few other cases, the code was using the same pointer variable to represent different sized arrays. For such cases, we added new variables for differently sized arrays so that all variables are used in a bound-consistent manner. Once the code compiled, we were able to run the executable and start the FTP server. We were prepared to debug any runtime crashes due to failed checks inserted by Checked C (this can be caused either by a real memory-safety violation in the code, or due to incorrect dynamic casts). However, we did not encounter any such cases.

Conclusion. Although this was only one case study, our experience showed that MSA annotation indeed helped. MSA roughly performed 58% of the work requires for end-to-end porting (250/426 edits). It inferred 123 annotations correctly which led to 250 edits, leaving 11 annotations for manual work, requiring 28 edits. In contrast, manually adding all the 112 annotations missed by 3C would have required much more effort. Further 148 edits were required to make the code compile; these edits were the easiest conceptually. We acknowledge that further work is required in this space to make a claim on end-to-end porting.

VII. Threats to Validity

One potential threat to validity is *data contamination*, if the Checked C versions of programs were part of the training data of the LLM. Given our use of LLMs from OpenAI, whose training data is not publicly known, there is no good way to completely rule it out. We note that Checked C versions of icecast, thttpd and vsftpd are indeed available publicly. However, there are no publicly available Checked C versions of lua and libarchive, to the best of our knowledge. The results across these benchmarks are largely consistent. Further, the LLM demonstrates poor knowledge of Checked C, unable to carry out the task without a background (**V0**, Section VI). Our contribution of whole-program transformations in a modular fashion stands, irrespective of data contamination (**V1** vs. MSA, Section VI). Also, Checked C code is extremely rare compared to other languages.

Another internal threat is our manual assessment of the correctness of annotations produced by MSA. We counter this by having multiple authors make independent assessment, then discussing to reach consensus. In some cases (Table I and Section VI-C), we validate correctness using the compiler, and found that our assessments were indeed correct.

An external threat to validity is that GPT4 can get updated over time by OpenAI, changing the performance of MSA. We also make no claim on the results that will be obtained with other LLMs.

VIII. Related Work

There are several approaches for memory safety. One set of tools use binary instrumentation to ensure safety of memory accesses [14], [15], [16]. These do not require recompilation, but suffer from high performance overheads. Overheads are lower for source-level instrumentation tools, at the cost of needing recompilation, but still high enough to prohibit production use [17], [18], [19]. Cyclone [3] is a safe dialect of C that extends pointers with labels (*ordinary*, *never-null*, and *fat*) and uses them to instrument safety checks. Deputy [2] provides bounded pointers and tagged unions. Checked C takes inspiration from this line of work and goes much further in building a production-ready compiler [20].

There has been a flurry of recent work in using LLMs to help improve the productivity of software developers. Our work builds along in this direction, while focusing specifically on memory safety through a safe C dialect. We are unaware of

prior LLM-based (or ML-based) work for this problem; and we have already presented a detailed comparion against 3C, the state-of-the-art symbolic tool for this problem.

LLMs have been extensively used for program synthesis [21], [22], [23], [24], [25], including both code generation (generating a program from a natural language description) and completion (automatically completing an incomplete code fragment). Recent auto-completion systems are even able to operate across files within a repository [26]. There have also been proposals that use LLMs to synthesize repair patches [27], [28], [29], surpassing the capabilities of current automated program repair tools. In the direction of software security, Pearce et al. [30] train their own GPT2 model to evaluate the efficacy of LLMs at generating and repairing software vulnerabilities. Their study is restricted to small "vulnerability patterns" that lead to well-known CWE/CVEs. In the space of LLMs for software engineering, our contribution on whole-program transformations, with the ability of making several correlated edits to a program using a simple task description, is novel. We refer the reader to survey papers [31], [32] for more work on leveraging LLMs for software engineering.

There is also a series of work that explores the synergy of LLMs with program analysis. Llift [33] combines LLMs with static analysis tools to identify use-before-initialization bugs in the Linux kernel. Lemur [34] uses LLMs to infer loop invariants that are validated by a symbolic verifier. However, Lemur is evaluated only on small benchmarks that fit in a single LLM prompt. Our whole program transformation framework and techniques should be applicable to loop invariant inference as well. Chakroborty et al. [35] develop techniques to rank LLM-generated candidate loop invariants. It is an interesting future work to investigate if similar rankers can be developed for ranking LLM-generated candidates for Checked C annotations.

IX. Conclusion

Large Language Models have demonstrated promising capabilities for software engineering tasks. Tools like Github Copilot are used by millions of developers. However, for reliability of the LLM-generated code, the developers are on their own. On the other side, formal verification tools provide strong guarantees but are difficult to deploy on a large scale. They require annotations, program refactorings, etc. to be effective. Combining LLM program comprehension capabilities with formal verification tools provides best of both the worlds. While tedious tasks of annotating and refactoring code can be delegated to LLMs, they need not be trusted—formal verification tools can do the validation. In this paper, we have presented a general framework to combine LLMs with formal verification tools, instantiated with the concrete problem of inferring memory safety annotations. On our benchmarks, we find that LLMs are able to reason about the code to generate annotations, refactoring the code as and when required. Still, they are outside the Trusted Computing Base as the Checked C compiler catches any LLM mistakes (statically or at runtime). We invite the community to further explore the synergy between formal methods and LLMs.

References

[1] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, "Checked c: Making c safe by extension," in *2018 IEEE Cybersecurity Development (SecDev)*, 2018, pp. 53–60.

[2] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula, "Dependent types for low-level programming," in *Proceedings of the 16th European Symposium on Programming*, ser. ESOP'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 520–535.

[3] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c," in *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '02. USA: USENIX Association, 2002, p. 275–288.

[4] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Measuring github copilot's impact on productivity," *Commun. ACM*, vol. 67, no. 3, p. 54–63, feb 2024. [Online]. Available: https://doi.org/10.1145/3633453

[5] A. Machiry, J. Kastner, M. McCutchen, A. Eline, K. Headley, and M. Hicks, "C to checked c by 3c," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, apr 2022. [Online]. Available: https://doi.org/10.1145/3527322

[6] J. Duan, Y. Yang, J. Zhou, and J. Criswell, "Refactoring the freebsd kernel with checked C," in *IEEE Secure Development, SecDev 2020, Atlanta, GA, USA, September 28-30, 2020*. IEEE, 2020, pp. 15–22. [Online]. Available: https://doi.org/10.1109/SecDev45635.2020.00018

[7] J. S. Foster, M. Fähndrich, and A. Aiken, "A theory of type qualifiers," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, ser. PLDI '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 192–203. [Online]. Available: https://doi.org/10.1145/301618.301665

[8] A. Ruef, L. Lampropoulos, I. Sweet, D. Tarditi, and M. Hicks, "Achieving safety incrementally with checked C," in *Principles of Security and Trust - 8th International Conference, POST 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, ser. Lecture Notes in Computer Science, F. Nielson and D. Sands, Eds., vol. 11426. Springer, 2019, pp. 76–98. [Online]. Available: https://doi.org/10.1007/978-3-030-17138-4_4

[9] P. Deligiannis, A. Lal, N. Mehrotra, and A. Rastogi, "Fixing rust compilation errors using LLMs," *CoRR*, vol. abs/2308.05177, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.05177

[10] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, "Supporting dynamic data structures on distributed-memory machines," *ACM Trans. Program. Lang. Syst.*, 1995.

[11] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," ser. PLDI '94, 1994.

[12] Checked C, "Benchmarks for evaluating checked C," https://github.com/microsoft/checkedc/wiki/Benchmarks-for-evaluating-Checked-C, 2019.

[13] CheckedC Wiki, https://github.com/checkedc/checkedc/wiki/C-Conversion-Tips, 2023.

[14] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Proceedings of the Summer 1992 USENIX Conference*, 1992, pp. 125–138.

[15] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 89–100. [Online]. Available: https://doi.org/10.1145/1250734.1250746

[16] M. Burrows, S. N. Freund, and J. L. Wiener, "Run-time type checking for binary programs," in *Proceedings of the 12th International Conference on Compiler Construction*, 2003.

[17] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, V. Sarkar, B. G. Ryder, and M. L. Soffa, Eds., 1994.

[18] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: type-safe retrofitting of legacy code," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 128–139. [Online]. Available: https://doi.org/10.1145/503272.503286

[19] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "Softbound: highly compatible and complete spatial memory safety for c,"

in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009*, M. Hind and A. Diwan, Eds. ACM, 2009.

[20] Checked C, "Checked C clang compiler repository," https://github.com/checkedc/checkedc-llvm-project, 2024.

[21] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.abq1158

[22] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," in *ICLR*, 2023.

[23] G. Poesia, A. Polozov, V. Le, A. Tiwari, G. Soares, C. Meek, and S. Gulwani, "Synchromesh: Reliable code generation from pre-trained language models," in *International Conference on Learning Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=KmtVD97J43e

[24] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, ser. MAPS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–10. [Online]. Available: https://doi.org/10.1145/3520312.3534862

[25] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3491101.3519665

[26] T. Liu, C. Xu, and J. McAuley, "Repobench: Benchmarking repository-level code auto-completion systems," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=pPjZIOuQuF

[27] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *ICSE*, 2023.

[28] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs? an evaluation on quixbugs," in *Proceedings of the Third International Workshop on Automated Program Repair*, ser. APR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 69–75. [Online]. Available: https://doi.org/10.1145/3524459.3527351

[29] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.

[30] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 2339–2356. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.10179420

[31] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE Computer Society, may 2023, pp. 31–53.

[32] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *arXiv:2308.10620*, 2023.

[33] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An LLM-integrated approach," in *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA*, 2024.

[34] H. Wu, C. Barrett, and N. Narodytska, "Lemur: Integrating large language models in automated program verification," in *The 3rd Workshop on Mathematical Reasoning and AI at NeurIPS'23*, 2023. [Online]. Available: https://openreview.net/forum?id=NxHl2SPhyT

[35] S. Chakraborty, S. K. Lahiri, S. Fakhoury, A. Lal, M. Musuvathi, A. Rastogi, A. Senthilnathan, R. Sharma, and N. Swamy, "Ranking LLM-generated loop invariants for program verification," in *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. [Online]. Available: https://openreview.net/forum?id=R7f5euZ9RA