Checked C: Making C Safe by Extension

Archibald Samuel Elliott, University of Washington
Andrew Ruef and Michael Hicks, University of Maryland
David Tarditi, Microsoft Research

Abstract—This paper presents Checked C, an extension to C designed to support spatial safety, implemented in Clang and LLVM. Checked C's design is distinguished by its focus on backward-compatibility, incremental conversion, developer control, and enabling highly performant code. Like past approaches to a safer C, Checked C employs a form of *checked pointer* whose accesses can be statically or dynamically verified. Performance evaluation on a set of standard benchmark programs shows overheads to be relatively low. More interestingly, Checked C introduces the notions of a *checked region* and *bounds-safe interfaces*.

1 Introduction

Vulnerabilities that compromise *memory safety* are at the heart of many attacks. Memory safety has two aspects. *Temporal safety* is ensured when memory is never used after it is freed. *Spatial safety* is ensured when any pointer dereference is always within the memory allocated to that pointer. *Buffer overruns*—a spatial safety violation—still constitute a frequent and pernicious source of vulnerability, despite their long history. During the period 2012–2018, buffer overruns were the source of 9.7% to 18.4% of CVEs reported in the NIST vulnerability database [1], with the highest numbers occurring in 2017. During that time, buffer overruns were the leading single cause of CVEs.

As discussed in depth in Section 2, several efforts have attempted to make C programs safe. Static analysis tools [2], [3], [4] aim to find vulnerabilities pre-deployment, but may miss bugs, have trouble scaling, or emit too many alarms. Security mitigations, such as CFI [5] and DEP [6], can mute the impact of vulnerabilities by making them harder to exploit, but provide no guarantee; e.g., data leaks and mimicry attacks may still be possible. Several efforts have aimed to provide spatial safety by adding run-time checks; these include CCured [7], Softbound [8], and ASAN [9]. The added checks can add substantial overhead and can complicate interoperability with legacy code when pointer representations are changed. Lower overhead can be achieved by reducing safety, e.g., by checking only writes, or ignoring overruns within a memory region (e.g., from one stack variable to another, or one struct field to another). In the end, no existing approach is completely satisfying.

This paper presents a new effort towards achieving a spatially-safe C that we call *Checked C*. Checked C borrows many ideas from prior safe-C efforts but ultimately differs in that its design focuses on allowing incremental conversion while balancing control, interoperability, and high performance. Technically speaking, Checked C's design has three key features. First, all pointers in Checked C are represented as in normal C—no changes to pointer layout are imposed. This eases interoperability. Second, the legal boundaries of pointed-to memory are specified explicitly; the goal here is to enhance human readability and maintainability while supporting efficient compilation and running times. Checked C supports pointers to single objects, arrays, and NUL-terminated arrays. The types and bounds are used by the compiler to either prove that an access is safe, or else to insert a run-time bounds check when such a proof is too difficult. Programmers can use dynamic checks and annotations to convince the compiler to eschew unnecessary checks in performance-critical code.

Finally, Checked C is designed to support incremental porting from legacy C. Programs may consist of a mix of checked and legacy pointers, and fully ported code can be annotated as within a *checked region*, which can be held *blameless for any spatial safety violation*. This guarantee is made possible by restricting any use of unchecked pointers and casts within the region. To allow existing unchecked code to be accessed by checked regions and with checked pointers, Checked C allows unchecked code to be annotated with *bounds-safe interfaces*. These describe the expected behavior and requirements of the code and can be added to parameters and return values of function declarations/definitions, function and record types, and global variables. Such interfaces support modular porting and use of legacy libraries. In the Checked C universe, programmers can add safety with each use of a checked pointer, and then extend the safety guarantee by expanding the scope of checked regions. At every step, they enjoy a working software artifact. Ultimately, a fully-ported program is assuredly safe, and in the meantime scrutiny can be focused on any unchecked regions.

We are implementing Checked C as an extension to the LLVM compiler infrastructure. Preliminary results are promising. On a standard benchmark suite, Checked C adds an average run-time overhead of 8.6%, with about 90% of the code in checked regions. Code changes were modest: 17.5% of benchmark lines of code were changed, and most ($> 80\%$, on average) were trivial to perform.

Checked C is under active and ongoing development. We are porting more programs to evaluate its utility. We are developing a tool to rewrite legacy programs semi-automatically. We are also improving Checked C's analysis, and formalizing a proof of its safety guarantee. Checked C's code and in-depth specification are available at https://github.com/Microsoft/checkedc.

2 Prior Work

There has been extensive research addressing out-of-bounds memory accesses in C [10]. The research falls into four categories:

C(-like) dialects, compiler implementations, static analyses, and security mitigations.

Safe C Dialects. Cyclone [11] and Deputy [12], [13] are type-safe dialects of C. Cyclone's key novelty is its support for GC-free temporal safety [14], [15]. Checked C differs from Cyclone by being backward compatible (Cyclone disallowed many legacy idioms) and avoiding pointer format changes (e.g., Cyclone used "fat" pointers to support arithmetic). Deputy keeps pointer layout unchanged by allowing a programmer to describe the bounds using other program expressions. Deputy incorporates the bounds information into the types of pointers by using dependent types. Deputy requires that values of all pointers stay in bounds so that they match their types. To enforce this invariant (and make type checking decidable), it inserts runtime checks before pointer arithmetic, not at memory accesses. Checked C uses separate annotations that describe bounds invariants instead of incorporating bounds into pointer types and inserts runtime checks at memory accesses.

Non C-based programming languages like D [16] and Rust [17] also aim to support safe, low-level systems-oriented programming. Legacy programs would need to be ported wholesale to take advantage of these languages, which could be a costly affair (but could be partially automated [18]).

Safe C implementations. Rather than use a new language, several projects have looked at new ways to implement legacy C programs so as to make them spatially safe. The *bcc* source-to-source translator [19] and the *rtcc* compiler [20] changed the representations of pointers to include bounds. The *rtcc*-generated code was 3 times larger and about 10 times slower. Fail-Safe C [21] changed the representation of pointers and integers to be pairs. Benchmarks were 2 to 4 times slower. CCured [7] employed a whole-program analysis for transforming programs to be safe. Its transformation involved changes to data layout (e.g., fat and "wild" pointers), which could cause interoperation headaches. Compilation was all-or-nothing: unhandled code idioms in one compilation unit could inhibit compilation of the entire program.

Safety can also be offered by the loader and run-time system. "Red zones", used by Purify [22], [23] are inserted before and after dynamically-allocated objects and between statically-allocated objects, where bytes in the red zone are marked as inaccessible (at a cost of 2 bits per protected byte). Red-zone approaches cannot detect out-of-bounds accesses that occur entirely within valid memory for other objects or intra-object buffer overruns (a write to an array in a struct that overwrites another member of the struct). Checked C detects accesses to unrelated objects and intra-object overruns.

Checking that accesses are to the proper objects can be done using richer side data structures that track object bounds and by checking that pointer arithmetic stays in bounds [24], [25], [26], [27], [8], [28], [29]. Baggy Bounds Checking [27] provides a fast implementation of object bounds by reserving 1/n of the virtual address space for a table, where n is the smallest allowed object size and requiring object sizes be powers of 2. It increases SPECINT 2000 execution time by 60% and memory usage by 20%. SoftBound [8] tracks bounds information by using a hash table or a shadow copy of memory. It increases execution time for a set of benchmarks by 67% and average memory footprint by 64%. SoftBound can check only writes, in which case execution time increases by 22%.

There is also work on adding temporal safety with different memory allocation implementations, e.g., via conservative garbage collection (GC) [30] or regions [14], [15]. Checked C focuses on spatial safety both due to its importance at stopping code injection style attacks as well as information disclosure attacks. Temporal safety can be added by linking Checked C programs with a conservative GC; overcoming the cost of doing so (e.g., as done in Cyclone or Rust) is something we plan to investigate in the future.

Static analysis. Static analysis tools take source or binary code and attempt to find possible bugs, such as out-of-bounds array accesses, by analyzing the code. Commercial tools include CodeSonar, Coverity Static Analysis, HP Fortify, IBM Security AppScan, Klocwork, Microsoft Visual Studio Code Analysis for C/C++, and Polyspace Static Analysis [2], [31], [32]. Static analysis tools have difficulty balancing precision and performance. To be precise, they may not scale to large programs. While imprecision can aid scalability, it can result in false positives, i.e., error reports that do not correspond to real bugs. False positives are a significant problem [2]. As a result, tools may make unsound assumptions (e.g., inspecting only a limited number of paths through function [31]) but the result is they may also miss genuine bugs (false negatives). Alternatively, they may focus on supporting coding styles that avoid problematic code constructs, e.g., pointer arithmetic and dynamic memory allocation [4], [33], [34], [3]. Or, they may require sophisticated side conditions on specifications, i.e., as pre- and post-conditions at function boundaries, so that the analysis can be modular, and thus more scalable [35].

Checked C occupies a different design point than static analysis tools. It avoids problems with false positives by deferring bounds checks to runtime—in essence, it trades run-time overhead for soundness and coding flexibility.

Security mitigations. Security mitigations employ runtime-only mechanisms that detect whether memory has been corrupted or prevent an attacker from taking control of a system after such corruption. They include data execution prevention (DEP) [6], software fault isolation (SFI) [36] , address-space layout randomization (ASLR) [37], [38], stack canaries [39], shadow stacks [40], [41], and control-flow integrity (CFI) [5]. DEP, ASLR, and CFI focus on preventing execution of arbitrary code and control-flow modification. Stack protection mechanisms focus on protecting data or return addresses on the stack.

Checked C provides protection against data modification and data disclosure attacks, which the other approaches do not. For example, ASLR does not protect against data modification or data disclosure attacks. Data may be located on the stack adjacent to a variable that is subject to a buffer overrun; the buffer overrun can be be used reliably to overwrite or read the data. Shadow stacks do not protect stack-allocated buffers or arrays, heap data, and statically-allocated data. Chen et al. [42] show that data modification attacks that do not alter control-flow (Heartbleed is an example) pose a serious long-term threat.

3 CHECKED C

This section presents an overview of Checked C's main features.

3.1 Basics

The Checked C extension extends the C language with two additional *checked pointer types*: `_Ptr`<T> and `_Array_ptr`<

```
void
read_next(int *b, int idx, _Ptr<int>out) {
  int tmp = *(b+idx);
  *out = tmp;
}
```

Figure 1. Example use of _Ptr<T>

T>.[1] The _Ptr<T> type indicates a pointer that is used for dereference only and has no arithmetic performed on it, while _Array_ptr<T> supports arithmetic with bounds declarations provided in the type. The compiler dynamically confirms that checked pointers are valid when they are dereferenced. In blocks or functions specifically designated as *checked regions*, it imposes stronger restrictions on uses of unchecked pointers that could corrupt checked pointers, e.g., via aliases. We also extend C with *checked array types* (including NUL-terminated ones), by prefixing array dimensions with the _Checked keyword, as in **int** buf _Checked[10]. We expect a Checked C program to involve a mix of both checked and unchecked regions, and a mix of checked and unchecked pointer and array types.

3.2 Simple pointers

Using _Ptr<T> is straightforward: any pointer to an object that is only referenced indirectly, without any arithmetic or array subscript operations, can be replaced with a _Ptr<T>. For example, one frequent idiom in C programs is an out parameter, used to indicate an object found or initialized during parsing. Figure 1 shows using a _Ptr<**int**> for the out parameter. When this function is called, the compiler will confirm that it is given a valid pointer, or null. Within the function, the compiler will insert a null check before writing to out. Such null checks are elided when the compiler can prove they are unnecessary.

3.3 Arrays

The _Array_ptr<T> type identifies a pointer to an array of values. Prior safe-C efforts sometimes involve the use of *fat pointers*, which consist both of the actual pointer and information about the bounds of pointed-to memory. Rather than changing the run-time representation of a pointer in order to support bounds checking, in Checked C the programmer associates a *bounds expression* with each _Array_ptr<T>-typed variable and member to indicate where the bounds are stored. The compiler inserts a run-time check that ensures that deferencing an _Array_ptr<T> is safe (the compiler may optimize away the runtime check if it can prove it always passes). Bounds expressions consist of non-modifying C expressions and can involve variables, parameters, and struct field members. For bounds *on* members, the bounds can refer only to other members declared in the same structure. Bounds declarations on members are type-level program invariants that can be suspended temporarily when updating a specific struct object.

Figure 2 shows using _Array_ptr<T> with declared bounds as parameters to a function. In particular, the types of the dst and src arrays have bound expressions that refer to the function's other two respective parameters. In the body of the function, both src and dst are accessed as expected. The

1. We use the C++ style syntax for programmer familiarity, and precede the names with an underscore to avoid parsing conflicts in legacy libraries.

```
void append(
  _Array_ptr<char> dst : count(dst_count),
  _Array_ptr<char> src : count(src_count),
  size_t dst_count, size_t src_count)
{
  _Dynamic_check(src_count <= dst_count);
  for (size_t i = 0; i < src_count; i++) {
    if (src[i] == '\0') {
      break;
    }
    dst[i] = src[i];
  }
}
```

Figure 2. Example use of _Array_ptr<T>

```
size_t my_strlcpy(
 _Nt_array_ptr<char> dst: count(dst_sz - 1),
 _Nt_array_ptr<char> src, size_t dst_sz)
{
  size_t i = 0;
  _Nt_array_ptr<char> s : count(i) = src;
  while (s[i] != '\0' && i < dst_sz - 1) {
    dst[i] = s[i];
    ++i;
  }
  dst[i] = '\0';
  return i;
}
```

Figure 3. Example use of _Nt_array_ptr<T>

compiler inserts runtime checks before accessing the memory locations src[i] and dst[i]. The compiler optimizes away the check on src[i] because it can prove that i $<$ src_count, the size of src. The compiler also optimizes away the check dst[i] thanks to the _Dynamic_check placed outside the loop. Like an assert, this predicate evaluates the given condition and signals a run-time error if the condition is false; unlike assert, this predicate is not removed unless proven redundant. Here, its existence assures the compiler that i $<$ dst_count (transitively), so no per-iteration checks are needed.

There are two other ways to specify array bounds. The first is a *range*, specified by base and upper bound pointers. For example, the bounds expression on dst from Figure 2 could have been written bounds(dst,dst+dst_count). The second is an alternative to count called bytecount, which can be applied to either **void*** or _Array_ptr<**void**> types. A bytecount(n) expression applied to a pointer p would be equivalent to the range p through (**char** *)p+n. An example of this is given at the end of this section.

We can also annotate an array declaration as _Checked. Any implicit conversion of the array to a pointer value is treated as a _Array_ptr<T>. We add a restriction that all inner dimensions of checked arrays also be checked. We see both of these situations in Figure 4, shortly. Parameters with checked array types are treated as having _Array_ptr<T> types. If no bounds are declared, the bounds are implied by the array size, if it is known. T _Checked[] is a synonym for _Array_ptr<T>.

3.4 NUL-terminated Arrays

The _Nt_array_ptr<T> type identifies a pointer to an array of values (often **char**s) that ends with a NUL ('\0'). The bounds expression identifies the known-to-be-valid range of the pointer. This range can be expanded by reading the character *just past* the bounds to see if it is NUL.[2] If not, then the bounds can be expanded by one. Otherwise, the current bounds cannot be expanded, and only a `'\0'` may be written to this location. `_Nt_array_ptr`<T> types without explicit bounds default to bounds of `count(0)`, meaning that index 0 can be read safely. A `_Nt_array_ptr`<T> can be cast to a `_Array_ptr`<T> safely—for an `_Array_ptr`<T>, the character just past the bounds can not be read or written, which ensures the zero-termination invariant is maintained for any aliases.

An example use of `_Nt_array_ptr`<T> is given in Figure 3. It implements the `strlcpy` libC routine, which copies at most `dst_sz` characters from `src` to `dst`. We must alias `src` into the local variable `s` so that its count, `i`, can grow dynamically as the loop executes.

NUL-terminated arrays can be declared by using the keyword `_Nt_checked` instead of `_Checked`. An implicit conversion of an `_Nt_checked` array to a pointer produces an `_Nt_array_ptr`<T>.

3.5 Checked and unchecked regions

The safety provided by checked pointers can be thwarted by unsafe operations, such as writes to traditional pointers. For example, consider this variation of the code in Figure 1:

```
void more(int *b, int idx, _Ptr<int *>out) {
  int oldidx = idx, c;
  do {
    c = readvalue();
    b[idx++] = c;
  } while (c != 0);
  *out = b+idx-oldidx;
}
```

This function repeatedly reads an input value into `b` until a 0 is read, at which point it returns an updated `b` pointer via the checked `out` parameter. While we might expect that writing to `out` is safe, since it is a checked pointer, it will not be safe if the loop overflows `b` and in the process modifies `out` to point to invalid memory.

In a program with a mix of checked and unchecked pointers we cannot and should not expect complete safety. However, we would like to provide some assurance about which code is possibly dangerous, i.e., whether it could be the source of a safety violation. Code review and other efforts can then focus on that code. For this purpose Checked C provides *checked regions* of code. Such code is designated specifically at the level of a file (using a pragma), a function (by annotating its prototype), or a single block (by labeling that block, similar to an `asm` block).

An example checked block is shown in Figure 4. Outside of the `_Checked`-annotated region, an unchecked pointer is cast to a checked one. This cast is a potential source of problems (if `out` is bogus) and so would not be permitted in checked regions. Within the checked block, checked pointers declared inside and outside the block can be freely manipulated and the compiler performs the expected checks. The compiler also treats uses of the address-of operator `&` in a checked block as producing a checked pointer, not an unchecked one. When doing this to a `struct` field, the bounds are defined as the extent of that field.

2. This means that bounds of `count(n)` requires allocating `n+1` bytes.

```
void foo(int *out) {
  _Ptr<int> ptrout;
  if (out != (int *)0) {
    ptrout = (_Ptr<int>)out; // cast OK
  } else { return; }
  _Checked {
    int b _Checked[5][5];
    for (int i = 0; i < 5; i++) {
      for (int j = 0; j < 5; j++) {
        b[i][j] = -1; // access safe
    } }
    *ptrout = b[0][0];
  }
}
```

Figure 4. Example `_Checked` block, and `_Checked` array

```
size_t fwrite(void   *p : byte_count(s*n),
              size_t s,
              size_t n,
              FILE   *st : itype(_Ptr<FILE>));
```

Figure 5. Standard library checked interface

In general, within a checked region both null and bounds checks on checked pointers are employed as usual, but additional restrictions are also imposed. In particular, explicit casts to checked pointer types are disallowed, as are reads from and writes to unchecked pointers. Checked regions may neither use varargs nor K&R-style prototypes. All of these restrictions are meant to ensure that the entire execution of a checked region is *spatially safe*.

Checked blocks allow for incremental conversion at a finer level of granularity than functions. A porting effort can identify that some code in a function can be made to use checked pointers entirely, while other code in the function can use a mixture of checked and unchecked pointers, as well as casts that can't be cheaply proven to be safe. Over time, the scope of the checked blocks can increase until they encompass the entire function.

In a sense, we can think of each use (e.g., a dereference) of a checked pointer as being contained with a tiny checked region consisting only of that use. This use is safe if the pointer itself is valid. `_Checked` regions expand the scope of such a safety guarantee to include not just the use of the pointer, but all code around it within the region. We have formalized a simple language that constitutes the core of Checked C and proved that this property holds. We are continuing to flesh out this formalization to include more tricky elements of Checked C's type system.

3.6 Bounds safe interfaces

Checked C permits ascribing checked types and bounds expressions to unchecked functions, members, and global variables via *bounds safe interfaces*. This allows checked regions to use legacy libraries and for the interactions to be checked. As an example, the type we give to the `fwrite` standard library function is shown in Figure 5. The first argument to the function is the target buffer whose size (in bytes) is given by the second and third arguments. The presence of a bounds expression indicates that an argument with type `T*` should have a checked type `_Array_ptr`<T> ascribed to it. For `p` this is `_Array_ptr`<

`void`>. The final argument is a `FILE` pointer, which is ascribed the checked type given by the `itype` annotation.

The type used during type checking depends on whether the function is called from a checked or unchecked region. For a checked region, it is the ascribed the checked type. For an unchecked region, an argument can have the unchecked or unchecked type. The type checker tries both types when checking the call. If the checked type is used, the argument must meet the requirements of a parameter's bounds expression also.

Bounds-safe interfaces may also be used to give checked types to as-yet unconverted code. As a project is converted, some files can be left alone, but have their externally visible signatures converted so that invoking their functions or using the types in checked blocks is allowed. For example, callers in a checked region could use the interface in Figure 5 to provide checked pointer types as parameters, while callers in an unchecked region can provide parameters with either unchecked or checked types.

3.7 Restrictions and limitations

Checked C is work in progress and currently has several limitations. First, to ensure that checked pointers are valid by construction, we require that checked pointer variables and structs/arrays containing checked pointers are initialized when they are declared. In addition, heap-allocated memory that contains checked pointers (like a struct or array of checked pointers) or is pointed to by a `_Nt_array_ptr`<T> must use `calloc` to ensure safe initialization. We plan to employ something akin to Java's *definite initialization* analysis to relax this requirement.

Second, we disallow taking the address of variables/**`struct`** members with bounds, variables used in bounds expressions, and members used in member bounds expressions. Such pointers could be used to subvert the validity of bounds checks.

Third, `_Array_ptr`<T> values can be dereferenced following essentially arbitrary arithmetic; e.g., if `x` is an `_Array_ptr`< **`int`**> we could dereference it via `*(x+y-n+1)` and the compiler will insert any needed checks to ensure the access is legal. However, *updates* to `_Array_ptr`<T> variables are currently more limited. The bounds for a variable are declared when the variable is declared. It is possible, however, that a variable may need different bounds at different points in the program. For example, we might like to replace the loop in Figure 2 with:

```
size_t i = 0;
for ( ; i < src_count; i++) {
  if (*src == '\0') break;
  *dst = *src;
  src++; dst++;
}
```

The problem is that the bounds declared for `src` are tantamount to the range (`src`,`src+src_count`). This means that updating `src` to `src+1` would invalidate them, as the upper bound would be off by one. We would like to declare `src` to have new bounds before entering the loop, such as

```
(src - i, src + src_count - i)
```

We plan to support flow-sensitive declarations of bounds, so that variables can have different bounds at different program points.

Finally, the design for checking statically that declared bounds are valid is incomplete. We could fall back on a combination of static and dynamic checking if necessary. We describe the state of our implementation in the next section.

4 Implementation

This section briefly describes our current implementation.

Compiler Implementation. We have been implementing Checked C as an extension to the Clang/LLVM compiler. The extension is enabled as a flag passed to Clang. We extended the C grammar [43] to support bounds declarations, the new `_Ptr`<T>, `_Array_ptr`<T>, and T `_Checked[N]` types, and adding `_Checked` or `_Unchecked` annotations to blocks and functions. We chose reserved identifiers so that they will not conflict with identifiers in existing code.

We include a set of *checked headers* which ascribe bounds-safe interfaces to C standard library functions. These are used in lieu of the standard headers when the Checked C flag is present.

In order to support Checked C's new types, we extended Clang's type representation and type checker. We added a pointer kind discriminator to Clang's compile-time pointer type representation, and a "checked" flag to Clang's compile-time array type representation.

Checked C's bounds expressions provide a static description of the bounds on a pointer. We check statically that the subexpressions of a bounds expression are *non-modifying expressions*: they do not contain any assignment, increment or decrement operators, or function calls. This ensures that using the expressions at bounds checks does not cause unexpected side-effects.

Checked C performs inference to compute a bounds expression that conservatively describes the bounds on a pointer-typed expression. Inference uses bounds expressions normalized into `bounds(`l, u`)` form. The inferred bounds are used to check memory accesses using the value of the pointer expression.

For pointer variables, the inferred bounds are the declared bounds. For pointer arithmetic expressions, the inferred bounds are those of the pointer-typed subexpression. When taking the address of a **`struct`**'s member (`&p->f`), the bounds are those of the particular field. On the other hand, the address of an array element retains the bounds of the whole array. For example, the bounds of **`int`** `x[5]` are

```
bounds(x, x+5*sizeof(int))
```

as are the bounds of `&x[3]`, rather than (say)

```
bounds(x+3*sizeof(int),x+4*sizeof(int))
```

The compiler must statically ensure that bounds declarations are valid after assignments and initialization. This requires two steps. First, a *subsumption check* confirms that assigning to a variable (an lvalue, more generally) meets the bounds required of pointers stored in the variable (lvalue). The required pointer bounds must be within (subsumed) by the inferred bounds of the right-hand expression. (Subsumption also applies to initialization and function parameter passing.) This check allows assignment to narrow, but not to widen, the bounds of the right-hand side value. Determining the required bounds is generally straightforward. In the simplest case, the bounds for pointers stored in a variable (lvalue) are directly declared, e.g., for a local variable or function parameter. For assignments to struct members, uses of struct members within the bounds expression for the member are replaced with an appropriate struct access, For example, given

```
struct S {
  int len;
  _Array_ptr<int> buf : count(len);
};
```

the required bounds for an assignment to `a.buf` are `a.len`.

Second, the compiler must ensure bounds expressions are still valid after a statement modifies a variable used in a bounds expression. For example, in Figure 3 the bounds of `s` is `count(i)`, but `i` is modified in a loop that iterates over `s` looking for a NUL terminator. For `_Array_ptr`<T> types, the modification is justified by subsumption: The updated bounds can be narrowed but not widened. For `_Nt_array_ptr`<T> types, we can widen the bounds by 1 byte if we know that the rightmost byte is `'\0'`, e.g., due to a prior check, as is the case in Figure 3.

At the moment subsumption checking is rather primitive. Some subsumption checks for bounds declarations that could be statically proven are not. Currently, the static analysis can only reason about bounds expressions that are syntactically equivalent (modulo constant-folding and ignoring non-value changing operations) and bounds expressions that are constant-sized ranges (syntactically equivalent base expressions +/- constant offsets). The main issue is the need to perform a more sophisticated dataflow analysis (at the Clang AST level) to gather and consider relevant facts about relationships between variables (such as equalities and inequalities).

The compiler complains when it cannot (dis)prove a subsumption check in checked code. In our experimental evaluation, we manually review the warnings. We insert the code in an `_Unchecked` block (for checks that are trivially obvious) or perform a dynamic subsumption check with `_Dynamic_bounds_cast` (which eliminates the error).

We have designed but not yet implemented the analysis that checks assignments to variables used in bounds declarations. For our experimental evaluation, we verified by hand that such assignments do not happen.

The Checked C compiler inserts run-time checks into the evaluation of lvalue expressions whose results are derived from checked pointers and that will be used to access memory. The code for these checks is handed to LLVM, which we allow to remove checks if it can prove they will always pass. In general, such checks are the only source of Checked C run-time overhead (aside from programmer use of `_Dynamic_check`).

Before any `_Ptr`<T> accesses the compiler inserts a run-time check that the pointer is non-null. Before any `_Array_ptr`<T> accesses the compiler inserts a non-null check followed by the required bounds check computed from the inferred bounds. The compiler does not perform any range checks during pointer arithmetic (unless the arithmetic accesses memory).

5 Preliminary Experimental Evaluation

We (mostly) converted two existing C benchmarks as an initial evaluation of the consequences of porting code to Checked C. We quantify both the changes required for the code to become checked, and the overhead imposed on compilation, running time, and executable size.

We chose the Olden [44] and Ptrdist [45] benchmark suites, described in Table 1, because they are specifically designed to test pointer-intensive applications, and they are the same benchmarks used to evaluate both Deputy [12] and CCured [7]. We did not convert *bc* from the Ptrdist suite and *voronoi* from the Olden suite for lack of time.

The evaluation results are presented in Table 2. These were produced using a 12-Core Intel Xeon X5650 2.66GHz, with 24GB of RAM, running Red Hat Enterprise Linux 6. All compilation and benchmarking was done without parallelism. We ran each benchmark 21 times with and without the Checked C changes using the test sizes from the LLVM versions of these benchmarks. We report the median; we observed little variance.

Code Changes. On average, we modified around 17.5% of benchmark lines of code. Most of these changes were in declarations, initializers, and type definitions rather than in the program logic. In the evaluation of Deputy [13], the reported figure of lines changed ranges between 0.5% and 11% for the same benchmarks, showing they have a lower annotation burden than Checked C.

We modified the benchmarks to use checked blocks and the top-level checked pragma. We placed code that could not be checked because it used unchecked pointers in unchecked blocks. On average, about 9.3% of the code remained unchecked after conversion, with a minimum and maximum of 3.9% and 20.4%. The cause was almost entirely variable-argument `printf` functions.

We manually inspected changes and divided them into *easy* changes and *hard* changes. Easy changes include: replacing included headers with their checked versions; converting a $T*$ to a `_Ptr`<T>; adding the `_Checked` keyword to an array declaration; introducing a `_Checked` or `_Unchecked` region; adding an initializer; and replacing a call to `malloc` with a call to `calloc`. The remaining "hard" changes include changing a $T*$ to a `_Array_ptr`<T> and adding a bounds declaration, adding structs, struct members, and local variables to represent run-time bounds information, and code modernization.

In all of our benchmarks, we found the majority—more than 80% on average–of changes were easy. In six of the benchmarks, the only "hard" changes were adding bounds annotations relating to the parameters of `main`.

In three benchmarks—em3d, mst, and yacr2—we had to add intermediate structs so that we could represent the bounds on `_Array_ptr`<T>s nested inside arrays. In mst we also had to add a member to a struct to represent the bounds on an `_Array_ptr`<T>. In the first case, this is because we cannot represent the bounds on nested `_Array_ptr`<T>s, in the second case this is because we only allow bounds on members to reference other members in the same struct. In em3d and anagram we also added local temporary variables to represent bounds information. In yacr2 there are a lot of bounds declarations that are all exactly the same where global variables are passed as arguments, inflating the number of "hard" changes.

Running time overhead. The average run-time overhead introduced by added dynamic checks was 8.6%. In more than half of the benchmarks the overhead was less than 1%. We believe this to be an acceptably low overhead that better static analysis may reduce even further.

In all but two benchmarks—treadd and ft—the added overhead matches (is within 2%) or betters that of Deputy. For yacr2 and em3d, Checked C does substantially better than Deputy, whose overheads are 98% and 56%, respectively. Checked C's overhead betters or matches that reported by CCured in every case but ft.

Compile-time overhead. On average, the overhead added to compilation time by using Checked C is 24.3%. The maximum overhead is 83.1%, and the minimum is 4.9% faster than compiling with C. Across the benchmarks, there is an average 7.4% code size overhead from the introduction of dynamic checks. Ten of the programs have a code size increase of less than 10%.

Name	LoC	Description
bh	1,162	Barnes & Hut N-body force computation
bisort	262	Sorts using two disjoint bitonic sequences
em3d	476	Simulates electromagnetic waves in 3D
health	338	Simulates Columbian health-care system
mst	325	Minimum spanning tree using linked lists
perimeter	399	Perimeter of quad-tree encoded images
power	452	The Power System Optimization problem
treadd	180	Sums values in a tree
tsp	415	Estimates Traveling-salesman problem
voronoi	814	Voronoi diagram of a set of points
anagram	346	Generates anagrams from a list of words
bc	5,194	An arbitrary precision calculator
ft	893	Fibonacci heap Minimum spanning tree
ks	549	Schweikert-Kernighan partitioning
yacr2	2,529	VLSI channel router

Table 1

Compiler Benchmarks. Top group is the Olden suite, bottom group is the Ptrdist suite. Descriptions are from [44], [45]. We did not convert *voronoi* from the Olden suite and *bc* from the Ptrdist suite.

	Code Changes			Observed Overheads		
Name	*LM* %	*EM* %	*LU* %	*RT* ±%	*CT* ±%	*ES* ±%
bh	10.0	76.7	5.2	+0.2	+23.8	+6.2
bisort	21.8	84.3	7.0	0.0	+7.3	+3.8
em3d	35.3	66.4	16.9	+0.8	+18.0	-0.4
health	24.0	97.8	9.3	+2.1	+18.5	+6.7
mst	30.1	75.0	19.3	0.0	+6.3	-5.0
perimeter	9.8	92.3	5.2	0.0	+4.9	+0.8
power	15.0	69.2	3.9	0.0	+21.6	+8.5
treadd	17.2	92.3	20.4	+8.3	+83.1	+7.0
tsp	9.9	94.5	10.3	0.0	+47.6	+4.6
anagram	26.6	67.5	10.7	+23.5	+16.8	+5.1
ft	18.7	98.5	6.3	+25.9	+16.5	+11.3
ks	14.2	93.4	8.1	+12.8	+32.3	+26.7
yacr2	14.5	51.5	16.2	+49.3	+38.4	+24.5
Mean:	17.5	80.1	9.3	+8.6	+24.3	+7.4

Table 2

Benchmark Results. Key: *LM %*: Percentage of Source LoC Modified, including Additions; *EM %*: Percentage of Code Modifications deemed to be Easy; *LU %*: Percentage of Lines remaining Unchecked; *RT* ±*%*: Percentage Change in Run Time; *CT* ±*%*: Percentage Change in Compile Time; *ES* ±*%*: Percentage Change in Executable Size (`.text` section only). *Mean*: Geometric Mean.

6 Future Work

Developing a practical, industrial-strength language is a substantial effort. As such, we have several threads of ongoing work.

Automatically rewrite C programs to Checked C: We are in the process of porting more and larger programs to Checked C. To help, we have been developing a tool that uses a simple analysis (inspired by that of CCured [7]) to convert pointer types to use `_Ptr`<T>. We are working on extending this tool to support both the placement of casts and bounds safe interfaces, as well as inferring bounds for array accesses that will allow conversion to `_Array_ptr`<T> types as well.

Improve the checking of bounds declarations: A key design goal of Checked C is to keep static checking fast and predictable. To nevertheless support sophisticated static reasoning, we are working on an analysis that infers relational invariants and adds them as source-level assertions. These assertions can be easily proved by the static checker and then used to justify eliminating costly dynamic checks.

Supporting polymorphism with safe casts: Polymorphic data structures such as lists and trees often declare their contents with type `void*`, casting to/from that type when inserting/removing data. Casts are also used to implement "poor man's subtyping" for struct types. We are extending Checked C to support these and other forms of safe cast, for greater flexibility.

Formalization: We have mechanized a formalization of a core subset of Checked C and used it to prove that any violation of spatial safety is due, directly or indirectly, to code occurring an unchecked region; checked code cannot be blamed. We are working to expand this formalism to include features such dynamically sized arrays and flow-sensitive updates to array lengths due to pointer assignments or NUL checks.

7 Conclusion

We have presented *Checked C*, an extension to C to help ensure spatial safety. Checked C's design is focused on interoperability with legacy C, usability, and high performance. Any part of a program may contain, and benefit from, *checked pointers*. Such pointers are binary-compatible with legacy, unchecked pointers but have explicitly annotated and enforced bounds. Code units annotated as *checked regions* provide guaranteed safety: The code within may not use unchecked pointers or unsafe casts that could result in spatial safety violations. Checked C's *bounds-safe interfaces* provide checked types to unchecked code, which is useful for retrofitting third party and standard libraries. Together, these features permit incrementally adding safety to a legacy program, rather than making it an all-or-nothing proposition. Our implementation of Checked C as an LLVM extension enjoys good performance, with relatively low run-time and compilation overheads. It is freely available at https://github.com/Microsoft/checkedc and continues to be actively developed.

References

[1] "NIST vulnerability database," https://nvd.nist.gov, accessed May 17, 2017.

[2] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010.

[3] Mathworks, "Polyspace code prover: prove the absence of run-time errors in software," http://www.mathworks.com/products/polyspace-code-prover/index.html, 2016, accessed May 12, 2016.

[4] AbsOmt, "Astrée: Fast and sound runtime error analysis," http://www.absint.com/astree/index.htm, 2016, accessed May 12, 2016.

[5] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti, "Control-flow integrity," in *ACM Conference on Computer and Communications Security*, 2005.

[6] S. Andersen and V. Abella, "Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies," 2004.

[7] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[8] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for C," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[9] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, 2012.

[10] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

[11] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, , and Y. Wang, “Cyclone: A safe dialect of C,” in *USENIX Annual Technical Conference*. Monterey, CA: USENIX, 2002, pp. 275–288.

[12] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer, “SafeDrive: Safe and recoverable extensions using language-based techniques,” in *7th Symposium on Operating System Design and Implementation (OSDI'06)*. Seattle, Washington: USENIX Association, 2006.

[13] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula, “Dependent types for low-level programming,” in *Proceedings of European Symposium on Programming (ESOP '07)*, 2007.

[14] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, “Region-based memory management in Cyclone,” in *PLDI*, 2002.

[15] N. Swamy, M. Hicks, G. Morrisett, D. Grossman, and T. Jim, “Safe manual memory management in Cyclone,” *Sci. of Comp. Programming*, vol. 62, no. 2, pp. 122–144, Oct. 2006, special issue on memory management. Expands ISMM conference paper of the same name.

[16] dlang.org, “D,” http://dlang.org/, 2016, accessed May 13, 2016.

[17] Rust-lang.org, “Rust documentation,” https://www.rust-lang.org/documentation.html, 2016, accessed May 13, 2016.

[18] “C to rust translation, refactoring, and cross-checking,” https://c2rust.com/, 2018.

[19] S. C. Kendall, “Bcc: runtime checking for C programs,” in *USENIX Toronto 1983 Summer Conference*. Berkeley, CA, USA: USENIX Association, 1983.

[20] J. L. Steffen, “Adding run-time checking to the Portable C Compiler,” *Softw. Pract. Exper.*, vol. 22, no. 4, pp. 305–316, Apr. 1992.

[21] Y. Oiwa, “Implementation of the memory-safe full ANSI-C compiler,” in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2009.

[22] R. Hastings and B. Joyce, “Purify: Fast detection of memory leaks and access errors,” in *Proceedings of the Winter 1992 USENIX Conference*. Berkeley, CA, USA: USENIX Association, 1992, pp. 125–138.

[23] I. Unicom Systems, “Purifyplus,” http://unicomsi.com/products/purifyplus/, 2016, accessed May 6, 2016.

[24] R. W. M. Jones and P. H. J. Kelly, “Backwards-compatible bounds checking for arrays and pointers in C programs,” in *Third International Workshop on Automated Debugging*, ser. Linkoping Electronic Conference Proceedings, M. Kamkar and D. Byers, Eds. Linkoping University Electronic Press, May 1997, "http://www.ep.liu.se/ea/cis/1997/009/".

[25] H. Patil and C. Fischer, “Low-cost, concurrent checking of pointer and array accesses in C programs,” *Software: Practice & Experience*, vol. 27, no. 1, pp. 87–110, Jan. 1997.

[26] O. Ruwase and M. S. Lam, “A practical dynamic buffer overflow detector,” in *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, 2004.

[27] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, “Preventing memory error exploits with WIT,” in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[28] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, “Paricheck: An efficient pointer arithmetic checker for c programs,” in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[29] G. J. Duck and R. H. C. Yap, “Heap bounds protection with low fat pointers,” in *Proceedings of the 25th International Conference on Compiler Construction*, 2016.

[30] H.-J. Boehm and M. Weiser, “Garbage collection in an uncooperative environment,” *Softw. Pract. Exper.*, vol. 18, no. 9, pp. 807–820, Sep. 1988.

[31] W. R. Bush, J. D. Pincus, and D. J. Sielaff, “A static analyzer for finding dynamic programming errors,” *Softw. Pract. Exper.*, vol. 30, no. 7, pp. 775–802, Jun. 2000.

[32] P. Emanuelsson and U. Nilsson, “A comparative study of industrial static analysis tools,” *Electron. Notes Theor. Comput. Sci.*, vol. 217, pp. 5–21, Jul. 2008.

[33] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, “A static analyzer for large safety-critical software,” in *PLDI*, 2003.

[34] D. Delmas and J. Souyris, “Astrée: From research to industry,” in *Proceedings of the 14th International Conference on Static Analysis*, 2007.

[35] B. Hackett, M. Das, D. Wang, and Z. Yang, “Modular checking for buffer overflows in the large,” in *ICSE*, 2006.

[36] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient software-based fault isolation,” in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, 1993.

[37] P. Team, http://pax.grsecurity.net/docs/aslr.txt, 2001.

[38] Wikipedia, “Address space layout randomization,” https://en.wikipedia.org/wiki/Address_space_layout_randomization, 2016, accessed April 25, 2016.

[39] C. Cowan, C. Pu, D. Maiere, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks,” in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, 1998.

[40] A. Baratloo, N. Singh, and T. Tsai, “Transparent run-time defense against stack smashing attacks,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2000.

[41] T.-c. Chiueh and F.-H. Hsu, “RAD: A compile-time solution to buffer overflow attacks,” in *Proceedings of the The 21st International Conference on Distributed Computing Systems*, 2001.

[42] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, “Non-control-data attacks are realistic threats,” in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, 2005.

[43] ISO, “ISO/IEC 9899:2011 - Information Technology - Programming Languages - C (C11 standard),” Geneva, 2011.

[44] 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.*, vol. 17, no. 2, pp. 233–263, Mar. 1995.

[45] T. M. Austin, S. E. Breach, and G. S. Sohi, “Efficient detection of all pointer and array access errors,” *SIGPLAN Not.*, vol. 29, no. 6, Jun. 1994.