Checked C for Safety, Gradually

Andrew Ruef
University of Maryland

Archibald Samuel Elliott
University of Washington

Ian Sweet
University of Maryland

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, developer usability, 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. New to Checked C is the notion of a *checked region*. Inspired by the *blame theorem* from *gradual typing*, checked regions can be held blameless as the source of a safety violation, meaning it must have arisen from unchecked code. We formalize and prove this property using the Coq proof assistant. To assist programmers in migrating legacy code to Checked C, we have implemented a porting tool that introduces the use of checked pointers, where safe. Experiments on standard benchmarks and some legacy programs show that Checked C generates efficient code, and that the porting tool is useful.

1 Introduction

Vulnerabilities that compromise *memory safety* are at the heart of many devastating 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 2012–2016, buffer overruns were the source of 9.7% to 18.4% of CVEs reported in the NIST vulnerability database [36], with the highest numbers occurring in 2016. During that time, buffer overruns were the leading single cause of CVEs.

Spatial safety violations commonly arise when programming low-level, performance critical code in C and C++. While a type-safe language disallows such violations [49], using one is impractical when low-level control is needed. Building on research from projects such as Cyclone [26] and Deputy [57], modern languages like Rust [42] and Go [21] provide a promising balance of safety and performance, but to use them requires programmer retraining and extensive rewrites of legacy code.

As discussed in depth in Section 7, several efforts have attempted to make C programs safe. Static analysis tools [2, 6, 29] aim to find vulnerabilities pre-deployment, but may miss bugs, have trouble scaling, or emit too many alarms. Security mitigations, such as W⊕X [45] and CFI [1], 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 [34], Softbound [33], and ASAN [44]. The added checks can add substantial overhead and can complicate interoperability with legacy code if 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 interoperability, developer usability, and enabling highly performant code. Checked C and legacy C can coexist, so developers are able to port legacy code incrementally. This approach does allow for defects and vulnerabilities in non-converted regions of the program. However, taking inspiration from work on *gradual typing* [30, 46, 53], Checked C gives developers a way to distinguish "checked" from "unchecked" regions. The former can be held blameless as the source of any safety violation, so software assurance attention can be focused on the latter.

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. As an example, consider the following code declarations:

```
size_t dst_count;
_Array_ptr<char> dst : count(dst_count);
```

The `_Array_ptr<char>` type is a Checked C type for a bounds-checked array, and the `count` annotation indicates how the bounds should be computed. In this case `dst`'s bounds are stored in the variable `dst_count`, but other specifications, such as pointer ranges, are also possible. Checked C also

Draft: 17 November 2017

has a _Ptr<*T*> type for pointers to single *T* values, and a _Nt_array_ptr<*T*> type for pointers to NUL (zero) terminated arrays. Checked type information is used by the compiler to either prove that an access is safe, or else to insert a bounds check when such a proof is too difficult. Programmers can also use annotations to help the compiler safely avoid adding unnecessary checks in performance-critical code.

Finally, Checked C supports the concept of designated *checked regions* of code. Within these regions, use of unchecked pointers is essentially disallowed, so the above-mentioned checks are sufficient to ensure that *execution is spatially safe*: no failure will occur within the region assuming its checked pointers are well formed (i.e., they have not been corrupted through prior execution of unchecked code). In short, in the parlance of gradual typing, "checked code cannot be blamed" [53] for a spatial safety violation. We have formalized the core features of Checked C and proved it satisfies the blame theorem, mechanizing most of the proof using the Coq proof assistant.

Several prior efforts have eschewed annotations, citing the programmer cost of adding them to legacy code. However, in our experience programmers have a sense of the extents and invariants of memory objects and prefer to document and enforce them, but C gives them no easy mechanism to write them down. To assist in the process of updating legacy code, Checked C employs an automated tool to partially rewrite an application to use Checked C types. We believe this approach strikes the right balance: A best-effort analysis can be applied to the whole program to assist in porting, but once ported, a program's annotations ensure efficient checking and assist readability and maintainability. The rewriter uses a global, path-insensitive unification-based algorithm to infer when variables, structure fields, function parameters, and function return values might be converted to Checked C _Ptr<*T*> and _Array_ptr<*T*> types. It automatically rewrites the program to add the former types, and points to locations for the latter, at which the programmer can convert them by hand, adding needed bounds expressions. To avoid one unsafe pointer use forcing all transitive uses to be unchecked, the rewriter may insert casts, taking advantage of Checked C's ability to mix checked and unchecked code.

Contributions This paper makes four main contributions.

First, in Section 2, we present Checked C's design and its rationale, introducing its various features by example.

Second, in Section 3, we formalize the core ideas in the design of Checked C in a core calculus called CoreChkC. We show that, in the style of gradual typing, any misbehavior can be blamed on unchecked code—either it will misbehave directly, or could induce misbehavior in checked code.

Third, as described in Section 4, we have implemented Checked C as an extension to Clang and LLVM. Since Checked C is a backwards compatible superset of C, any project that compiles today with Clang and LLVM can compile with Checked C. As reported in Section 6, we converted most of the standard Olden and Ptrdist benchmark suites to use Checked C. On average, we modified 17.5% of the benchmark code so that 90.7% of it could be placed in checked regions. The mean run-time slowdown is 8.6%, which generally matches or betters Deputy [57] and CCured [34] on the same benchmarks.

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

Figure 1. Example use of _Ptr<*T*>

Finally, as described in Section 5 we have implemented a tool to automatically convert existing C programs to Checked C programs. This tool performs a whole-program, context- and flow-insensitive analysis to identify types that can be replaced with Checked C types, and automatically rewrites them. In about 35 minutes of work the rewriter was able to replace between 37% and 69% of C pointer types with _Ptr<*T*> types in six benchmark programs, comprising more than 290KLOC.

Checked C is under active and ongoing development, and available on the Internet at https://github.com/Microsoft/checkedc.

2 Checked C

This section presents an overview of Checked C.

2.1 Basics

The Checked C extension extends the C language with two additional *checked pointer types*: _Ptr<*T*>, _Array_ptr<*T*> and _Nt_array_ptr<*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*> and _Nt_array_ptr<*T*> support arithmetic with bounds declarations provided in the type. The latter requires NUL termination, which affords some flexibility on determining bounds. The compiler statically or dynamically confirms that checked pointers are valid when they are dereferenced. In blocks or functions designated as *checked code*, it imposes stronger restrictions to uses of unchecked pointers that could corrupt checked pointers, e.g., via aliases. We would expect a Checked C program to involve a mixed of both checked and unchecked code, and a mix of checked and unchecked pointer types.

2.2 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*>.

[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 buf_copy(
  _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;
    else { dst[i] = src[i]; }
  }
}
```

Figure 2. Example use of _Array_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. Null checks are elided when the compiler can prove they are unnecessary.

2.3 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 to support bounds checking, in Checked C the programmer associates a *bounds expression* with each _Array_ptr<*T*> type to indicate where the bounds are stored. The compiler proves that indexing an _Array_ptr<*T*> is safe or else inserts a run-time check. Bounds expressions consist of non-modifying C expressions and can involve variables, parameters, and struct field members.

Figure 2 shows using _Array_ptr<*T*> with declared bounds as parameters to a function. In particular, the types of dst and src have bound expressions that refer to the function's other two parameters. (On struct members, bounds declarations may refer to the same struct's fields.) In the body of the function, both src and dst are accessed as expected. Bounds checks on src are elided because the compiler can prove that i ≤ src_count, the size of src. Checks on dst are elided thanks to the _Dynamic_check placed outside the loop. Like an assert, this predicate signals a run-time error if the condition is false, but it is only removed if 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 bounds 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

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

Figure 3. Example use of _Nt_array_ptr<*T*>

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 and any auto-promoted address to that array is treated as a checked _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.

2.4 NUL-terminated Arrays

The _Nt_array_ptr<*T*> type identifies a pointer to an array of values (often chars) 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; as an _Array_ptr<*T*> the character just past the bounds can no longer be read or written, thus preserving the zero-termination invariant for any aliases.

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

2.5 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 i, _Ptr<int *>out) {
  int oldi = i, c;
```

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

```
int *out;
_Checked void foo(void) {
  _Ptr<int> ptrout = 0;
  _Unchecked {
    if (out != (int *)0) {
      ptrout = (_Ptr<int>)out; // cast OK
    } else { return; }
  }
  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. `_Unchecked` and `_Checked` regions (and array)

```
  do {
    c = readvalue();
    b[i++] = c;
  } while (c != 0);
  *out = b+i-oldi;
}
```

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 isolate 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 introduces the notion of *checked code regions*. 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). Explicitly labeled *un*checked regions may also appear within checked ones.

Figure 4 shows a checked function `foo`, which references unchecked pointer `out` within an explicitly labeled `_Unchecked` block. Without this label, the compiler would forbid this cast since it is a potential source of problems (i.e., if `out` was bogus). 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 declarations of and casts, reads, and writes from unchecked pointer types are disallowed. Checked regions may neither use varargs nor K&R-style prototypes. These restrictions are meant to ensure that the entire execution of a checked region is *spatially safe*. This means that assuming checked pointers have been constructed properly (in particular, they have not been corrupted by the execution of unchecked code prior to entering the checked region), no safety violations will occur due to dereferencing a pointer into illegal memory. Section 3 makes this guarantee precise, and proves that it holds.

```
size_t fwrite(
  const void * ptr : byte_count(size*nmemb),
  size_t size, size_t nmemb,
  FILE * stream : itype(_Ptr<FILE>));
```

Figure 5. Standard library checked interface

Checked C also permits ascribing checked types to unchecked functions. We use this feature in a set of *checked headers* for the C standard library. As an example, the type we give to the `fwrite` 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 final argument is a `FILE` pointer whose type depends on whether it is being called from checked or unchecked code. For the former, the type is given by the `itype` annotation, indicating it is expected to be a checked pointer. For the latter, it is the "normal" type of the argument.

2.6 Restrictions and Limitations

Checked C's design currently imposes several restrictions.

First, to ensure that checked pointers are valid by construction, we require that checked pointer variables be 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, at least somewhat.

Second, `_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*> values are currently more limited. For example, we might like to replace the loop in Figure 2 with the following:

```
for (size_t i = 0; 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`), which would mean that updating `src` to `src`+1 would invalidate them, as the upper bound would be off by one. At the moment, this sort of arithmetic is allowed by assigning `src` and `dst` to temporary variables; updating these variables is OK because the bounds would be in terms of `src` and `dst`, which would not

Mode	m	::=	$c \mid u$
Word types	τ	::=	$\texttt{int} \mid \texttt{ptr}^m \omega$
Types	ω	::=	$\tau \mid \texttt{struct}\ T \mid \texttt{array}\ n\ \tau$
Expressions	e	::=	$n^\tau \mid x \mid \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2$
		|	$\texttt{malloc@}\omega \mid (\tau)e$
		|	$e_1 + e_2 \mid \&e{\rightarrow}f$
		|	$*e \mid *e_1 = e_2 \mid \texttt{unchecked}\ e$
Structdefs	D	$\in$	$T \rightharpoonup fs$
Fields	fs	::=	$\tau\ \texttt{f} \mid \tau\ \texttt{f}; fs$

Figure 6. CoreChkC Syntax

change. We plan to support *flow-sensitive bounds* so that, for example, the update `src++` would update `src`'s bounds to (`src`,`src+src_count-i`).

Finally, some elements of our static analysis for confirming safe usage are designed but not fully implemented. We elaborate on these in Section 4.

3 Formalism: CoreChkC

This section presents a formal language CoreChkC that models the essence of Checked C. The language is designed to be simple but nevertheless highlight Checked C's key features: checked pointers; checked code blocks, which are prevented from using unchecked pointers and certain casts; and unchecked code blocks, which may manipulate pointers as they wish. After presenting the syntax, semantics, and type system of CoreChkC, we state and prove its key guarantee: Checked code cannot be blamed for a spatial violation.

3.1 Syntax

The syntax of CoreChkC is presented in Figure 6. Types τ classify word-sized objects while types ω also include multi-word objects. The type $\texttt{ptr}^m\omega$ types a pointer, where m identifies its *mode*: mode c identifies a Checked C safe pointer, while mode u represents an unchecked pointer. In other words $\texttt{ptr}^c\tau$ is a checked pointer type `_Ptr<`τ`>` while $\texttt{ptr}^u\tau$ is an unchecked pointer type $\tau*$. Multiword types ω include `struct` records, and arrays of type τ having size n, i.e., $\texttt{ptr}^c\texttt{array}\ n\ \tau$ represents a checked array pointer type `_Array_ptr<`τ`>` with bounds n. We assume `structs` are defined separately in a map D from struct names to their constituent field definitions.

Programs are represented as expressions e; we have no separate class of program statements, for simplicity. Expressions include (unsigned) integers n^τ, local variables x, which are introduced by let-bindings $\texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2$; there is no type annotation on variable x because it can be inferred from context. Constant integers n are annotated with type τ to indicate their intended type. As in an actual implementation, pointers in our formalism are represented as integers. Annotations help formalize type checking and the safety property it provides; they have no effect on the semantics except when τ is a checked pointer, in which case they facilitate null and bounds checks. Local variables can only hold word-sized objects, so all `structs` can only be accessed by pointers.

Checked pointers are constructed using $\texttt{malloc@}\omega$; for simplicity, we do not consider numeric arguments to `malloc`, but just include the type. Thus, `malloc@int` produces a pointer of type $\texttt{ptr}^c\texttt{int}$ while $\texttt{malloc@(array 10 int)}$ produces a pointer of type $\texttt{ptr}^c\texttt{(array 10 int)}$. Unchecked pointers can only be produced by the cast operator, $(\tau)e$, e.g., by doing $(\texttt{ptr}^u\texttt{int})\texttt{malloc@int}$. Casts can also be used to coerce between integer and pointer types and between different multi-word types.

Pointers are read via the $*$ operator, and assigned to via the = operator. To read or write `struct` fields, a program can take the address of that field and read or write that address, e.g., $x{\rightarrow}f$ is equivalent to $*(\&x{\rightarrow}f)$. To read or write an array, the programmer can use pointer arithmetic to access the desired element, e.g., $x[i]$ is equivalent to $*(x + i)$.

By default, CoreChkC expressions are assumed to be checked. Expression e in $\texttt{unchecked}\ e$ is unchecked, giving it additional freedom: Checked pointers may be created via casts, and unchecked pointers may be read or written.

Design Notes. CoreChkC leaves out many interesting C language features. We do not include an operation for freeing memory, since this paper is concerned about spatial safety, not temporal safety. CoreChkC models statically sized arrays but supports dynamic indexes; supporting dynamic sizes is interesting but less important compared to the complexity it adds the formalism. Making `ints` unsigned simplifies handling pointer arithmetic. We do not model control operators or function calls, whose addition would be straightforward. Function calls $f(e')$ can be modeled by $\texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2$, where we can view x as function f's parameter, e_2 as its body, and e_1 as its actual argument. Calls to unchecked functions from checked code can thus be simulated by having an $\texttt{unchecked}\ e$ expression for e_2. We chose to make checked mode the default in the formalism, but making it unchecked would have been equally easy, as would be the addition of $\texttt{checked}\ e$ expression.

3.2 Semantics

Figure 7 shows a portion of the small-step operational semantics for CoreChkC expressions; the full semantics is given in Appendix A. The figure defines judgment $H; e \longrightarrow^m H; r$. Here, H is a *heap*, which is a partial map from integers (representing pointer addresses) to type-annotated integers n^τ. m is the *mode* of evaluation, which is either c for checked mode, or u for unchecked mode. Finally, r is a *result*, which is either an expression e, `Null` (indicating a null pointer dereference), or `Bounds` (indicating an out-of-bounds array access). An unsafe program execution occurs when the expression reaches a *stuck* state — the program is not an integer n^τ, and yet no rule applies. Notably, this could happen if trying to

$$
\begin{array}{llll}
\text{Heap} & H & \in & \mathbb{Z} \rightharpoonup \mathbb{Z} \times \tau \\
\text{Result} & r & ::= & e \mid \texttt{Null} \mid \texttt{Bounds} \\
\text{Contexts} & E & ::= & \_ \mid \texttt{let}\ x\ =\ E\ \texttt{in}\ e \\
 & & \mid & E + e \mid n + E \\
 & & \mid & \&E{\rightarrow}f \mid (\tau)E \\
 & & \mid & *E \mid *E = e \mid *n = E \\
 & & \mid & \texttt{unchecked}\ E
\end{array}
$$

C-Exp

$$\frac{e = E[e_0] \qquad m = mode(E) \qquad H; e_0 \rightsquigarrow H'; e_0' \qquad e' = E[e_0']}{H; e \longrightarrow^m H'; e'}$$

C-Halt

$$\frac{e = E[e_0] \qquad m = mode(E) \qquad H; e_0 \rightsquigarrow H'; r \qquad r = \texttt{Null} \vee r = \texttt{Bounds}}{H; e \longrightarrow^m H'; r}$$

$$
\begin{array}{ll}
mode(\_) & = c \\
mode(\texttt{unchecked}\ E) & = u \\
mode(\texttt{let}\ x\ =\ E\ \texttt{in}\ e) & = \\
\quad mode(E + e) & = \\
\quad mode(n + E) & = \\
\quad mode(\&E{\rightarrow}f) & = \\
\quad mode((\tau)E) & = \\
\quad mode(*E) & = \\
\quad mode(*E = e) & = \\
\quad mode(*n = E) & = mode(E)
\end{array}
$$

Figure 7. Semantics (partial)

dereference a pointer n that is actually invalid, i.e., $H(n)$ is undefined.

The semantics is defined in the standard manner using *evaluation contexts* E. We write $E[e_0]$ to mean the expression that results from substituting e_0 into the "hole" _ of context E. Rule C-Exp defines normal evaluation. It decomposes an expression e into a context E and expression e_0 and then evaluates the latter via $H; e_0 \rightsquigarrow H'; e_0'$, discussed below (and defined in the Appendix, Figure 10). The annotation m is the evaluation mode, which is restricted by the $mode(E)$ function, also given in Figure 7. The rule and this function ensure that when evaluation occurs within e in some expression unchecked e, then it does so in unchecked mode u; otherwise it may be in checked mode c. Rule C-Halt halts evaluation due to a failed null or bounds check.

The rules for the computation semantics $H; e_0 \rightsquigarrow H'; e_0'$ are straightforward. As mentioned above, the annotations τ on literals n^τ indicate the type the program has ascribed to n. When τ is a checked pointer, the rules use it to model bounds and null checks. For example, dereferencing n^τ where τ is $\texttt{ptr}^c(\texttt{array}\ l\ \tau_0)$ produces `Bounds` when $l = 0$ and `Null` when $n = 0$. The semantics updates l, the array length, when performing checked pointer arithmetic. When a type annotation is not a checked pointer, the semantics ignores it. For example, addition $n_1^{\tau_1} + n_2^{\tau_2}$ ignores τ_1 and τ_2 when τ_1 is not a checked pointer, and simply annotates the result with it.

3.3 Typing

The typing judgment $\Gamma \vdash_m e : \tau$ says that expression e has type τ under environment Γ when in mode m. Heap H and struct map D are implicit parameters of the judgment; they do not appear because they are invariant in derivations. Unchecked expressions are always checked in mode u, otherwise we may use either mode.

Γ maps variables x to types τ, and is used in rules T-Var and T-Let as usual. Rule T-Int ascribes type τ to literal n^τ when τ is `int` or an unchecked pointer type (so dereferencing is only possible in unchecked code, and failure there is an option), when n is 0 (and thus dereferencing it in checked

T-Var

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash_m x : \tau}$$

T-VConst

$$\frac{n^\tau \in \Gamma}{\Gamma \vdash_m n^\tau : \tau}$$

T-Let

$$\frac{\Gamma \vdash_m e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash_m e_2 : \tau}{\Gamma \vdash_m \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 : \tau}$$

T-Int

$$\frac{\tau = \texttt{int} \vee \tau = \texttt{ptr}^u\omega \vee n = 0 \vee \tau = \texttt{ptr}^c(\texttt{array}\ 0\ \tau')}{\Gamma \vdash_m n^\tau : \tau}$$

T-PtrC

$$\frac{\tau = \texttt{ptr}^c\omega \qquad \tau_0, \ldots, \tau_{j-1} = \text{types}(D, \omega) \qquad \Gamma, n^\tau \vdash_m H(n+k) : \tau_k\ 0 \le k < j}{\Gamma \vdash_m n^\tau : \tau}$$

T-Amper

$$\frac{\Gamma \vdash_m e : \texttt{ptr}^m\texttt{struct}\ T \qquad D(T) = \ldots; \tau_f\ f; \ldots}{\Gamma \vdash_m \&e{\rightarrow}f : \texttt{ptr}^m\tau_f}$$

T-BinopInt

$$\frac{\Gamma \vdash_m e_1 : \texttt{int} \qquad \Gamma \vdash_m e_2 : \texttt{int}}{\Gamma \vdash_m e_1 + e_2 : \texttt{int}}$$

T-Malloc

$$\frac{}{\Gamma \vdash_m \texttt{malloc@}\omega : \texttt{ptr}^c\omega}$$

T-Unchecked

$$\frac{\Gamma \vdash_u e : \tau}{\Gamma \vdash_m \texttt{unchecked}\ e : \tau}$$

T-Cast

$$\frac{m = c \ \Rightarrow\ \tau \neq \texttt{ptr}^c\omega\ (\text{for any } \omega) \qquad \Gamma \vdash_m e : \tau'}{\Gamma \vdash_m (\tau)e : \tau}$$

T-Deref

$$\frac{\Gamma \vdash_m e : \texttt{ptr}^{m'}\omega \qquad \omega = \tau \vee \omega = \texttt{array}\ n\ \tau \qquad m' = u \Rightarrow m = u}{\Gamma \vdash_m *e : \tau}$$

T-Index

$$\frac{\Gamma \vdash_m e_1 : \texttt{ptr}^{m'}(\texttt{array}\ n\ \tau) \qquad \Gamma \vdash_m e_2 : \texttt{int} \qquad m' = u \Rightarrow m = u}{\Gamma \vdash_m *(e_1 + e_2) : \tau}$$

T-Assign

$$\frac{\Gamma \vdash_m e_1 : \texttt{ptr}^{m'}\omega \qquad \Gamma \vdash_m e_2 : \tau \qquad \omega = \tau \vee \omega = \texttt{array}\ n\ \tau \qquad m' = u \Rightarrow m = u}{\Gamma \vdash_m *e_1 = e_2 : \tau}$$

T-IndAssign

$$\frac{\Gamma \vdash_m e_1 : \texttt{ptr}^{m'}(\texttt{array}\ n\ \tau) \qquad \Gamma \vdash_m e_2 : \texttt{int} \qquad \Gamma \vdash_m e_3 : \tau \qquad m' = u \Rightarrow m = u}{\Gamma \vdash_m *(e_1 + e_2) = e_3 : \tau}$$

Figure 8. Typing

mode would produce Null) or it has type $\texttt{ptr}^c(\texttt{array}\ 0\ \tau')$ (since dereferencing it would produce Bounds).

Rule T-PtrC ensures checked pointers of type $\texttt{ptr}^c\omega$ are consistent with the heap. This works by checking that the pointed-to memory has types consistent with ω. When doing this, we add n^τ to Γ to properly handle cyclic heap structures, per rule T-VConst. A key feature of T-PtrC is that it effectively confirms that all pointers reachable from the given one are consistent; it says nothing about other parts of the heap. For example, if some set of checked pointers is only reachable via unchecked pointers then we are not concerned whether the former are consistent, since they cannot be accessed (directly) from checked pointers.

Rules T-Amper and T-BinopInt are unsurprising. Rule T-Malloc produces checked pointers. Rule T-Unchecked introduces unchecked mode, relaxing access rules. Rule T-Cast enforces that checked pointers cannot be cast targets in checked mode.

Rules T-Deref and T-Assign type pointer accesses. These rules require unchecked pointers only be dereferenced in unchecked mode. Rule T-Index permits reading a computed pointer to an array, and rule T-IndAssign permits writing to one. These rules are not strong enough to permit updating a pointer to an array after performing arithmetic on it (though this is possible in our implementation). In general, Checked C's design permits overcoming such limitations through selective use of casts in unchecked code.

3.4 Metatheory

Our main formal result is that well-typed programs will never fail with a spatial safety violation that is due to a checked region of code, i.e., *checked code cannot be blamed.* This result is proved using two lemmas.

The first lemma, Progress, indicates that a well-typed program either is a value, can take a step (in either mode), or else is stuck in unchecked code. The latter is true if e only type checks in mode u, or its (unique) context E has mode u.

Lemma 3.1 (Progress). *If $\cdot \vdash_m e : \tau$ (under heap H) then one of the following holds:*

- *e is an integer n^τ*
- *There exists H', m', and r such that $H; e \longrightarrow^{m'} H'; r$ where r is either some e',* Null, *or* Bounds.
- *$m = u$ or $e = E[e'']$ and $mode(E) = u$ for some E, e''.*

The second lemma, Preservation, implies that if a well-typed program in checked mode takes a checked step then the resulting program is also well-typed in checked mode.

Lemma 3.2 (Preservation). *If $\Gamma \vdash_c e : \tau$ (under a heap H), and $\vdash \Gamma$, and $H; e \longrightarrow^c H'; r$ (for some H', r), then and $r = e'$ implies $H \rhd H'$ and $\Gamma \vdash_c e' : \tau$ (under heap H').*

Here we write $\vdash \Gamma$ to mean $\nexists n^\tau \in \Gamma$, i.e., Γ just contains mappings of variables to types. We write $H \rhd H'$ to mean that for all n^τ if $\cdot \vdash n^\tau : \tau$ under H then $\cdot \vdash n^\tau : \tau$ under H'. The proofs of both lemmas are by induction on the typing derivation. The Preservation proof is the most delicate, particularly ensuring $H \rhd H'$ despite the creation or modification of cyclic data structures.

With these results we can prove a *blame theorem* in the style defined in the gradual typing literature [30, 46, 53]. In particular, the theorem "well-typed code can't be blamed" [53] indicates that the statically typed part of a mixed-typing program cannot be blamed for execution getting stuck; only corruption by or execution of the dynamically typed component can inhibit progress. For Checked C, the same situation holds, respectively, for checked and unchecked code.

Theorem 3.3 (Checked code cannot be blamed). *Suppose $\cdot \vdash_c e : \tau$ (under heap H) and there exists H_i, m_i, and e_i for $1 \le i \le k$ such that $H; e \longrightarrow^{m_1} H_1; e_1 \longrightarrow^{m_2} \ldots \longrightarrow^{m_k} H_k; e_k$. If $H_k; e_k$ is stuck then the source of the issue is unchecked code.*

Proof. Suppose $\cdot \vdash_c e_k : \tau$ (under heap H_k). By Progress, the only way the $H_k; e_k$ can be stuck is if $e_k = E[e'']$ and $mode(E) = u$; i.e., the term's redex is in unchecked code. Otherwise $H_k; e_k$ is not well typed, i.e., $\cdot \not\vdash_c e_k : \tau$ (under heap H_k). As such, one of the steps of the evaluation was in unchecked code, i.e., there must exist some i where $1 \le i \le k$ and $m_i = u$. This is because, by Preservation, a well-typed program in checked mode that takes a checked step always leads to a well-typed program in checked mode. □

This theorem means that a code reviewer can focus on unchecked code regions, trusting that checked ones are safe.

We have mostly mechanized our paper proofs using the Coq proof assistant.

4 Implementation

We have implemented Checked C as an extension to the Clang/LLVM 5.0 compiler, comprising about 16.5k LoC added or changed (per git diff). This section describes the various changes we made. Our fork of Clang is available online at https://github.com/Microsoft/checkedc-clang.

4.1 Overview

We extended the Clang C front-end to support the changes described in Section 2; the LLVM IR's analyses and optimizers were unchanged. We extended the C grammar to support checked pointers, bounds expressions, and (un)checked blocks, and made corresponding changes to Clang's data structures and static type checker. The compiler enforces the restrictions described in Sections 2 and 3 by statically confirming that array pointer bounds are correctly ascribed and maintained, and by inserting run-time bounds and non-null checks on pointer accesses, which are optimized away by LLVM if they can be proved redundant.

4.2 Checking Bounds

Ensuring that bounds expressions are correct requires two steps. First, the *subsumption check* confirms that assigning to an lvalue expression meets the bounds required of pointers stored at the lvalue, i.e. the required pointer bounds are subsumed by the rvalue bounds. (Subsumption also applies to initialization and function parameter passing.) This check allows assignment to narrow, but not to widen, the bounds of the assigned value. Determining these bounds is generally straightforward. In the simplest case, the bounds for pointers stored at an lvalue are directly declared, e.g., for a local variable or function parameter. When taking the address of a `struct`'s member (`&p->f`), the pointer 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))`.

Second, the compiler ensures bounds expressions are still valid after a statement modifies a variable in that 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, so some checks that could be statically proven are not. The main issue is the need to perform a more sophisticated dataflow analysis (at the Clang AST level) to gather and consider relevant facts. The compiler warns when it cannot (dis)prove a subsumption check and we manually review the warnings. For spurious ones, we insert the code in an `_Unchecked` block, or else perform a dynamic subsumption check with `_Dynamic_bounds_cast`.

4.3 Run-time Checks

The compiler inserts run-time checks into the evaluation of lvalue expressions whose lvalue is derived from a checked pointer and whose lvalue will be used to access memory. For example, `*p` produces an lvalue; the run-time check is part of the evaluation of `*p`. These checks are added to the AST, which allows LLVM's optimizers to remove them if it can prove they will always pass.

Before any `_Ptr<T>` accesses the compiler inserts a check that the pointer is non-null. Before any `_Array_ptr<T>` or `_Nt_array_ptr<T>` access the compiler inserts a non-null check followed by the required bounds check computed from the inferred bounds. The compiler does not perform any bounds checks during pointer arithmetic. Programmers can insert dynamic checks (per Figure 2) via `_Dynamic_check` and `_Dynamic_bounds_cast`; these, too, may be optimized away.

The compiler should also disallow arithmetic on a checked pointer if (a) that pointer is null, or (b) the arithmetic would overflow, since both operations could produce a bogus pointer. We have not implemented these checks yet, but doing so should be straightforward. The lack of these checks should not negatively impact our experimental comparison in Section 6. Closely related systems Deputy [57] and CCured [34] lack the overflow check, too, and null checks on pointer arithmetic should be inexpensive because they are easily optimized. E.g., the null check on `for` loop-guard `*p` would make redundant the null check on `p++`.

5 Automatic Porting

Porting legacy code to use Checked C's features can be time consuming. To assist the process, we developed a source-to-source translator called *checked-c-convert* that discovers safely-used pointers and rewrites them to be checked. This section describes the tool; it is evaluated in Section 6.2.

5.1 Conversion tool design and overview

checked-c-convert aims to be sound while also producing edits that are minimal and unsurprising. A rewritten program should be recognizable by the author and it should be usable as a starting point for both the development of new features and additional porting. A particular challenge is to preserve syntactic structure of the program. Previous, similar analyses rarely interact well with the preprocessor (e.g., they consider macro expansions rather than the original macro) and sometimes require combining multiple source files into one, prior to analysis. These choices are problematic for us: We prefer to rewrite one file at a time (perhaps taking into account whole-program knowledge) and preserve the definition and use of macros, and other formatting, in the source code.

The *checked-c-convert* tool is implemented as a clang libtooling application. It traverses a program's AST to generate constraints based on pointer usage, solves those constraints, and rewrites the program by promoting some declared pointer types to be checked, and inserting some casts. The tool operates on post-preprocessed code, but has sufficient location information to be able to rewrite the original source files. Moreover, for macro expansions that have parameters, it considers all expansions of those parameters together, so as to be able to rewrite the original macro's definition. In effect, this produces a context- and flow-insensitive rewriting of macros, just as would occur for functions.

5.2 Constraint logic and solving

The basic approach to infer a *qualifier* q_i for each defined pointer variable i. Inspired by CCured's approach [34], qualifiers can be either PTR, ARR and UNK, ordered as a lattice $PTR < ARR < UNK$. Those variables with inferred qualifier PTR can be rewritten into `_Ptr<T>` types, while those with UNK are left as is. Those with the ARR qualifier are eligible

to have `_Array_ptr<T>` type. For the moment we only signal this fact in a comment and do not rewrite because we cannot always infer proper bounds expressions.

Qualifiers are introduced at each pointer declaration, i.e., parameter, variable, field, etc. Constraints are introduced as a pointer is used, and take one of the following forms:

$$
\begin{array}{l}
q_i = PTR \mid ARR \mid UNK \mid q_j \\
q_i = ARR \quad \Rightarrow \quad q_j = ARR \\
q_i = UNK \quad \Rightarrow \quad q_j = UNK \\
\neg(q_i = PTR \mid ARR \mid UNK)
\end{array}
$$

This constraint language is quite similar to that of CCured [34]. An expression that performs arithmetic on a pointer with qualifier q_i, either via + or `[]`, introduces a constraint $q_i = ARR$. Assignments between pointers introduce aliasing constraints of the form $q_i = q_j$. Casts introduce implication constraints based on the relationship between the sizes of the two types. If the sizes are not comparable, then both constraint variables in an assignment-based cast are constrained to UNK via an equality constraint. One difference from CCured is the use of negation constraints, which are used to fix a constraint variable to a particular Checked C type (e.g., due to a `_Ptr<T>` annotation). These would cause problems for CCured, as they might introduce unresolvable conflicts. But Checked C's allowance of checked and unchecked code can resolve them using explicit casts, as discussed below.

Constraints are generated for each file individually, at first. Then these constraints are "linked" when it can be determined that they refer to the same global definition. Solving the constraints produces a qualifier assignment $q_i = X$ where X is PTR, ARR, or UNK. Each qualifier is initially assigned to PTR. Solving the constraints works iteratively by propagating aliasing and equality constraints to UNK first; then aliasing, equality and implication constraints involving UNK; then aliasing, implication and equality constraints to ARR. Like CCured, this algorithm runs in linear time proportional to the number of pointer variables in the program.

One problem with unification-based analysis is that a single unsafe use might "pollute" the constraint system by introducing an equality constraint to UNK that transitively constrains unified qualifiers to UNK as well. For example, casting a `struct` pointer to a `unsigned char` buffer to write to the network would cause all transitive uses of that pointer to be unchecked. The tool takes advantage of Checked C's ability to mix checked and unchecked pointers to solve this problem. In particular, constraints for each function are solved locally, using separate qualifier variables for each external function's declared parameters. If a function declaration's parameters are lower in the lattice order than the definition's (e.g., the caller passes a `_Ptr<int>` to a function that requires a `int*`), we insert a cast at the call site. This cast makes evident to the programmer the potential risk of the call, and can be fixed manually if the callee is conservatively misclassified by the tool.

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 [4, 41]. We were unable to convert *voronoi* from the Olden suite and *bc* from the Ptrdist suite using the current version of Checked C.

6 Empirical Evaluation

This section presents an evaluation of the Checked C compiler and porting tool, considering performance and efficacy.

6.1 Compiler evaluation

We 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 [41] and Ptrdist [4] 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 [57] and CCured [34]. We did not convert *bc* from the Ptrdist suite and *voronoi* from the Olden suite for lack of time, but plan to soon.

The evaluation results are presented in Table 2. Graphs of these results are given in Appendix B. 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			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 (see 6.1.1); *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.

6.1.1 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`. Hard changes are all other changes, including 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 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.

6.1.2 Observed Overheads

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.

On average, the compile-time overhead added by using Checked C is 24.3%. The maximum overhead is 83.1%, and the minimum is 4.9% faster than compiling with C.

We also evaluated code size overhead, by looking at the change in the size of `.text` section of the executable. This excludes data that might be stripped, like debugging information. 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%.

6.2 Porting Tool Evaluation

We also evaluated the efficacy of our porting tool. To do so, we ran it on six programs and libraries and recorded how many pointer types the rewriter converted and how many casts were inserted. We chose these programs as they represent legacy, low level libraries that are used in commodity systems and frequently in security-sensitive contexts.

Table 3 contains the results. The value in the `_Ptr`<*T*> column indicates the number of `_Ptr`<*T*> added to the program that replace standard C pointers. These are re-written at the location they are declared. After investigation, there are usually two reasons that a pointer cannot be replaced with a `_Ptr`<*T*>: either some arithmetic is performed on the pointer,

Program	# of *	%/# _Ptr	Arr.	Unk.	Casts(% Calls)	LOC
zlib 1.2.8	649	62%/406	5%/36	31%/207	174 (22%)	6220
sqlite 3.18.1	32781	40%/13330	2%/739	57%/18712	25949 (35%)	134788
libarchive 3.3.1	20292	51%/10533	1%/218	47%/9541	6383 (17%)	80182
lua 5.3.4	4271	45%/1942	1%/76	52%/2253	407 (4%)	14585
libtiff 4.0.6	8687	37%/3278	2%/240	59%/5169	2119 (14%)	57091
vsftpd 3.0.3	2035	69%/1418	1%/32	28%/585	448 (9%)	15048

Table 3. Number of pointer types converted. The # of * column represents the number of pointer types in the program. The Arr and Unk columns represent constraints where the rewriter determined that the access into the pointer was via indexing (Arr) or that the constraints can't be captured by the rewriter (Unk) due to casts, assignment to a non-zero literal, or some other operation. The Casts column represents the number of casts inserted, compared to the percentage of call sites that were re-written to include a cast.

or it is passed as a parameter to a library function for which a bounds-safe interface does not exist. The table also indicates the versions of each program as computed with cloc and the number of casts inserted, compared to the percentage of call sites re-written to include casts.

6.3 Additional Porting Experience

After using the porting tool on vsftpd, we spent a few days on porting it further. It makes heavy use of pointers, and we find that all three of our categories of checked pointer come into heavy use. There are few idioms it uses that we cannot support, apart from the expected low-level interactions with the I/O subsystem. We detail our experience further in Appendix C.

7 Related work

There has been extensive research addressing out-of-bounds memory accesses in C [49]. The research falls into 4 categories: languages, implementations, static analysis, and security mitigations.

Safe languages. Cyclone [26] and Deputy [13, 57] are type-safe dialects of C. Cyclone's key novelty is its support for GC-free temporal safety [22, 48]. 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. Checked C builds on this, but make bounds checking a first-class part of the language. Deputy incorporates the bounds information into the types of pointers by using dependent types. This makes type checking hard to understand. 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. Checked C uses separate annotations that describe bounds invariants instead of incorporating bounds into pointer types and inserts runtime checks only at memory accesses.

Like Cyclone, programming languages like D [17] and Rust [42] aim to support safe, low-level systems-oriented programming without requiring GC. Go [21] and C# [32] target a similar domain. Legacy programs would need to be ported wholesale to take advantage of these languages, which could be a costly affair.

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 [28] and the *rtcc* compiler [47] changed the representations of pointers to include bounds. The *rtcc*-generated code was 3 times larger and about 10 times slower. Fail-Safe C [37] changed the representation of pointers and integers to be pairs. Benchmarks were 2 to 4 times slower. CCured [34] 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. Our rewriting algorithm is inspired by CCured's analysis with the important differences that (a) not every pointer need be made safe, and (b) the output is not a step in compilation, but programmer-maintainable source code.

Safety can also be offered by the loader and run-time system. "Red zones", used by Purify [25, 51] are inserted before and after dynamically-allocated object 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 stack frames 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.

Similar tools include Bounds Checker [31], Dr. Memory [9, 18], Intel Inspector [14], Oracle Solaris Studio Code Analyzer [38], Valgrind Memcheck [35, 52], Insure++ [39], and AddressSanitizer (ASAN) [44]. ASAN is incorporated into the LLVM and GCC compilers. It tracks the state of 8-byte chunks in memory. It increases SPEC CPU program execution time by 73% when checking reads and writes and 26% when only checking writes. SPEC CPU2006 average memory usage is 3.37 times larger. Light-weight Bounds Checking [24] uses a two-level table to reduce memory overhead.

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 [3, 19, 27, 33, 40, 43, 56]. Baggy Bounds Checking [3] 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 [33] 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%. For libraries that cannot be recompiled, wrapper functions must be provided that update metadata. Checked C only requires that checked headers be provided.

There is also work on adding temporal safety with different memory allocation implementations, e.g., via conservative garbage collection [8] or regions [22, 48]. Checked C focuses on spatial safety both due to its importance at stopping code injection style attacks as well as information disclosure attacks, though temporal safety is important and we plan to investigate it 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 [6, 10, 20]. 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 [6]. As a result, tools may make unsound assumptions (e.g., inspecting only a limited number of paths through function [10]) 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 [2, 7, 16, 29]. 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 [23].

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. In addition, Checked C avoids complicated specifications on functions. For example, a modular static analysis might have required the code in Figure 2 to include that `src_count` ≤ `dst_count` as a function pre-condition. While this constraint is not particularly onerous, some specifications can be. In Checked C, such side conditions are unnecessary; instead, soundness ensured by occasional dynamic checks.

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), software fault isolation (SFI) [54] , address-space layout randomization (ASLR) [50, 55], stack canaries [15], shadow stacks [5, 12], and control-flow integrity (CFI) [1]. 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. [11] show that data modification attacks that do not alter control-flow pose a serious long-term threat. The Heartbleed attack illustrates the damage possible.

Gradual Typing. As mentioned in Section 3.4 our theorem "checked code cannot be blamed" takes inspiration from the *blame theorem* in the gradual typing literature [30, 46, 53]. In that setting types are checked at function call boundaries on first-order data; checks on higher-order functions are delayed until the function is called. In our setting, analogous checks on pointer validity are delayed until a pointer is dereferenced. Hence the precise source of a failure (e.g., which unchecked region) is not tracked directly. Finding an efficient way of doing this would be interesting future work.

8 Summary

This paper presented Checked C, an extension to C to help enforce spatial safety. Checked C's design is focused on interoperability with legacy C, usability, and efficiency. Checked C's novel notion of *checked regions* ensures that "checked code cannot be blamed" for a safety violation; we have proved this property for an idealized calculus CoreChkC, mostly mechanizing the proof using the Coq proof assistant. Our

implementation of Checked C as a Clang/LLVM extension enjoys good performance. To assist in incrementally strengthening legacy code, we have developed a porting tool for automatically rewriting code to use checked pointers.

Checked C is an ongoing project, with code freely available on the Internet at https://github.com/Microsoft/checkedc. We are actively working to strengthen our static checker to endorse more safe coding idioms, and to improve runtime check elimination. We are also actively improving our porting tool. In the longer term we plan to support temporal safety checking.

Acknowledgments

We would like to thank Jijoong Moon and Wonsub Kim from Samsung for their assistance with the implementation of the Checked C compiler and the manual conversion of several benchmarks.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*. ACM, New York, NY, USA, 340–353. https://doi.org/10.1145/1102120.1102165

[2] AbsOmt. 2016. Astrée: Fast and sound runtime error analysis. http://www.absint.com/astree/index.htm. (2016). Accessed May 12, 2016.

[3] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing Memory Error Exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP '08)*. IEEE Computer Society, Washington, DC, USA, 263–277. https://doi.org/10.1109/SP.2008.30

[4] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. *SIGPLAN Not.* 29, 6 (June 1994), 290–301. https://doi.org/10.1145/773473.178446

[5] Arash Baratloo, Navjot Singh, and Timothy Tsai. 2000. Transparent Run-time Defense Against Stack Smashing Attacks. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '00)*. USENIX Association, Berkeley, CA, USA, 21–21. http://dl.acm.org/citation.cfm?id=1267724.1267745

[6] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[7] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A Static Analyzer for Large Safety-critical Software. *SIGPLAN Not.* 38, 5 (May 2003), 196–207. https://doi.org/10.1145/780822.781153

[8] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exper.* 18, 9 (Sept. 1988), 807–820.

[9] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 213–223. http://dl.acm.org/citation.cfm?id=2190025.2190067

[10] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. 2000. A Static Analyzer for Finding Dynamic Programming Errors. *Softw. Pract. Exper.* 30, 7 (June 2000), 775–802. https://doi.org/10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H

[11] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-control-data Attacks Are Realistic Threats. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14 (SSYM'05)*. USENIX Association, Berkeley, CA, USA, 12–12. http://dl.acm.org/citation.cfm?id=1251398.1251410

[12] Tzi-cker Chiueh and Fu-Hau Hsu. 2001. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *Proceedings of the The 21st International Conference on Distributed Computing Systems (ICDCS '01)*. IEEE Computer Society, Washington, DC, USA. http://dl.acm.org/citation.cfm?id=876878.879316

[13] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. 2007. Dependent Types for Low-Level Programming. In *Proceedings of European Symposium on Programming (ESOP '07) (Lecture Notes in Computer Science)*, Vol. 4421. Springer-Verlag, Heidelberg, 520–535.

[14] Intel Corporation. 2016. Intel Inspector. https://software.intel.com/en-us/intel-inspector-xe. (2016). Accessed May 6, 2016.

[15] Crispin Cowan, Calton Pu, Dave Maiere, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7 (SSYM'98)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1267549.1267554

[16] David Delmas and Jean Souyris. 2007. Astrée: From Research to Industry. In *Proceedings of the 14th International Conference on Static Analysis (SAS'07)*. Springer-Verlag, Berlin, Heidelberg, 437–451. http://dl.acm.org/citation.cfm?id=2391451.2391480

[17] dlang.org. 2016. D. http://dlang.org/. (2016). Accessed May 13, 2016.

[18] Dr. Memory. 2016. Dr. Memory: Memory Debugger for Windows, Linux, and Mac. http://www.drmemory.org/. (2016). Accessed May 6, 2016.

[19] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, New York, NY, USA, 132–142. https://doi.org/10.1145/2892208.2892212

[20] Pär Emanuelsson and Ulf Nilsson. 2008. A Comparative Study of Industrial Static Analysis Tools. *Electron. Notes Theor. Comput. Sci.* 217 (July 2008), 5–21. https://doi.org/10.1016/j.entcs.2008.06.039

[21] golang.org. 2016. The Go Programming Language. https://golang.org/. (2016). Accessed May 13, 2016.

[22] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based Memory Management in Cyclone. In *PLDI*.

[23] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. 2006. Modular Checking for Buffer Overflows in the Large. In *ICSE*.

[24] Niranjan Hasabnis, Ashish Misra, and R. Sekar. 2012. Light-weight Bounds Checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 135–144. https://doi.org/10.1145/2259016.2259034

[25] Reed Hastings and Bob Joyce. 1992. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference*. USENIX Association, Berkeley, CA, USA, 125–138.

[26] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference*. USENIX, Monterey, CA, 275–288.

[27] Richard W. M. Jones and Paul H. J. Kelly. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging (Linkoping Electronic Conference Proceedings)*, Miriam Kamkar and D. Byers (Eds.). Linkoping University Electronic Press. "http://www.ep.liu.se/ea/cis/1997/009/".

[28] Samuel C. Kendall. 1983. Bcc: runtime checking for C programs. In *USENIX Toronto 1983 Summer Conference*. USENIX Association, Berkeley, CA, USA.

[29] Mathworks. 2016. 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.

[30] Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-language Programs. In *POPL*.

[31] MicroFocus. 2016. DevPartner. http://www.borland.com/en-GB/Products/Software-Testing/Automated-Testing/Devpartner-Studio. (2016). Accessed May 6, 2016.

[32] Microsoft Corporation. 2016. C# Programming Guide. https://msdn.microsoft.com/en-us/library/67ef8sbd.aspx. (2016). Accessed May 13, 2016.

[33] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

[34] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 3 (2005), 477–526.

[35] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[36] NVDB [n. d.]. NIST vulnerability database. https://nvd.nist.gov. ([n. d.]). Accessed May 17, 2017.

[37] Yutaka Oiwa. 2009. Implementation of the Memory-safe Full ANSI-C Compiler. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/1542476.1542505

[38] Oracle Corporation. 2016. Oracle Solaris Studio. http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html. (2016). Accessed May 6, 2016.

[39] Parasoft. 2016. Memory Error Detection. https://www.parasoft.com/capability/memory-error-detection/. (2016). Accessed May 6, 2016.

[40] Harish Patil and Charles Fischer. 1997. Low-cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Software: Practice & Experience* 27, 1 (Jan. 1997), 87–110.

[41] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. 1995. Supporting Dynamic Data Structures on Distributed-memory Machines. *ACM Trans. Program. Lang. Syst.* 17, 2 (March 1995), 233–263. https://doi.org/10.1145/201059.201065

[42] Rust-lang.org. 2016. Rust Documentation. https://www.rust-lang.org/documentation.html. (2016). Accessed May 13, 2016.

[43] Olatunji Ruwase and Monica S. Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*. Internet Society, Reston, VA, USA, 159–169. http://www.internetsociety.org/doc/practical-dynamic-buffer-overflow-detector.

[44] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 28–28. http://dl.acm.org/citation.cfm?id=2342821.2342849

[45] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-space Randomization. In *CCS*. W⊕X protection is discussed in Section 1.1.

[46] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages.

[47] Joseph L. Steffen. 1992. Adding Run-time Checking to the Portable C Compiler. *Softw. Pract. Exper.* 22, 4 (April 1992), 305–316. https://doi.org/10.1002/spe.4380220403

[48] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2006. Safe Manual Memory Management in Cyclone. *Sci. of Comp. Programming* 62, 2 (Oct. 2006), 122–144. Special issue on memory management. Expands ISMM conference paper of the same name.

[49] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 48–62. https://doi.org/10.1109/SP.2013.13

[50] PaX Team. 2001. http://pax.grsecurity.net/docs/aslr.txt. (2001).

[51] Inc. Unicom Systems. 2016. PurifyPlus. http://unicomsi.com/products/purifyplus/. (2016). Accessed May 6, 2016.

[52] Valgrind. 2016. Valgrind. http://valgrind.org/. (2016). Accessed May 6, 2016.

[53] Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can'T Be Blamed. In *ESOP*.

[54] Robert Wahbe, Steven Lucco, Thoma E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*. ACM, New York, NY, USA, 203–216. https://doi.org/10.1145/168619.168635

[55] Wikipedia. 2016. Address space layout randomization. https://en.wikipedia.org/wiki/Address_space_layout_randomization. (2016). Accessed April 25, 2016.

[56] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. 2010. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS '10)*. ACM, New York, NY, USA, 145–156. https://doi.org/10.1145/1755688.1755707

[57] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. 2006. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *7th Symposium on Operating System Design and Implementation (OSDI'06)*. USENIX Association, Seattle, Washington.

Heap	H	$\in$	$\mathbb{Z} \rightharpoonup \mathbb{Z} \times \tau$
Result	r	::=	$e \mid \texttt{Null} \mid \texttt{Bounds}$
Contexts	E	::=	$\_ \mid \texttt{let}\ x\ =\ E\ \texttt{in}\ e$
		|	$E + e \mid n + E$
		|	$\&E \rightarrow f \mid (\tau)E$
		|	$*E \mid *E = e \mid *n = E$
		|	$\texttt{unchecked}\ E$

Figure 9. Semantics Definitions

A Full operational semantics

Figure 10 defines the full operational semantics for CoreChkC. The syntax of the language is given in Figure 6 (5), auxiliary definitions are given in Figure 9, and a discussion of the main judgment $H; e \longrightarrow^m H; r$ is given in Section 3.2. Here we carefully describe the rules with prefix E-, which define the core computation semantics $H; e_0 \rightsquigarrow H'; e_0'$. The semantics is implicitly parameterized by struct map D.

Rule E-Binop produces an integer n_3 that is the sum of arguments n_1 and n_2. When n_1 is a non-zero checked pointer to an array and n_2 is an `int`, result n_3's type annotation is annotated as a pointer to an array with its bounds suitably updated.[3] Otherwise, for non checked array pointers, n_3's type is just the same as n_1's type. (Checked array pointers whose value is zero are handled below.)

Rules E-Deref and E-Assign check the bounds of checked array pointers: the length l must be positive for the dereference to be legal. The rule permits the program to proceed for non-checked or non-array pointers (but the type system will forbid them).

Rule E-Amper takes the address of a `struct` field, according to the type annotation on the pointer, as long the pointer is not zero or not checked.

Rule E-Malloc allocates a checked pointer by finding a string of free heap locations and initializing each to 0, annotated to the appropriate type. Here, types(D, ω) returns n types, where these are the types of the corresponding memory words; e.g., if ω is a `struct` then these are the types of its fields (looked up in D), or if ω is an array τ of length k, then we will get back k τ's.

Rule E-Let uses a substitution semantics for local variables; notation $e[x \mapsto n^\tau]$ means that all occurrences of x in e should be replaced with n^τ.

Rule E-Unchecked returns the result of an unchecked block.

Rules with prefix X- describe failures due to bounds checks and null checks (on checked pointers). These are analogues to the E-Assign, E-Deref, E-Binop, and E-Amper cases. The first two rules indicate a bounds violation for size-zero array pointers. The next two indicate an attempt to dereference a null pointer. The last two indicate an attempt to construct a checked pointer from a null pointer via field access or pointer arithmetic.

[3] Here, $l - n_2$ is natural number arithmetic: if $n_2 > l$ then $l - n_2 = 0$.

E-Binop	$H; n_1^{\tau_1} + n_2^{\tau_2}$	$\rightsquigarrow$	$H; n_3^{\tau_3}$	where $n_3 = n_1 + n_2$ $\tau_1 = \texttt{ptr}^c(\texttt{array}\ l\ \tau)\ \wedge\ n_1 \neq 0\ \Rightarrow$ $\tau_3 = \texttt{ptr}^c(\texttt{array}\ l'\ \tau)$ where $l' = l - n_2$ $\tau_1 \neq \texttt{ptr}^c(\texttt{array}\ l\ \tau)\ \Rightarrow\ \tau_3 = \tau_1$
E-Cast	$H; (\tau) n^{\tau'}$	$\rightsquigarrow$	$H; n^{\tau}$	
E-Deref	$H; *n^{\tau}$	$\rightsquigarrow$	$H; n_1^{\tau_1}$	where $n_1^{\tau_1} = H(n)$ $\tau = \texttt{ptr}^c(\texttt{array}\ l\ \tau')\ \Rightarrow\ l > 0$
E-Assign	$H; *n^{\tau} = n_1^{\tau_1}$	$\rightsquigarrow$	$H'; n_1^{\tau_1}$	where $H(n)$ defined $\tau = \texttt{ptr}^c(\texttt{array}\ l\ \tau')\ \Rightarrow\ l > 0$ $H' = H[n \mapsto n_1^{\tau_1}]$
E-Amper	$H; \&n^{\tau} \rightarrow f_i$	$\rightsquigarrow$	$H; n_0^{\tau_0}$	where $\tau = \texttt{ptr}^{m'}\texttt{struct}\ T$ $D(T) = \tau_1 f_1; \ldots; \tau_k f_k$ for $1 \leq i \leq k$ $m' \neq c\ \vee\ n \neq 0\ \Rightarrow\ n_0 = n + i\ \wedge\ \tau_0 = \texttt{ptr}^{m'}\tau_i$
E-Malloc	$H; \texttt{malloc@}\omega$	$\rightsquigarrow$	$H', n_1^{\texttt{ptr}^c\omega}$	where sizeof$(\omega) = k$ and $n_1 \ldots n_k$ are consecutive $n_1 \neq 0$ and $H(n_1) \ldots H(n_k)$ are undefined $\tau_1, \ldots, \tau_k = \text{types}(D, \omega)$ $H' = H[n_1 \mapsto 0^{\tau_1}] \ldots [n_k \mapsto 0^{\tau_k}]$
E-Let	$H; \texttt{let}\ x\ =\ n^{\tau}\ \texttt{in}\ e$	$\rightsquigarrow$	$H; e[x \mapsto n^{\tau}]$	
E-Unchecked	$H; \texttt{unchecked}\ n^{\tau}$	$\rightsquigarrow$	$H; n^{\tau}$	
X-DerefOOB	$H; *n^{\tau}$	$\rightsquigarrow$	$H; \texttt{Bounds}$	where $\tau = \texttt{ptr}^c(\texttt{array}\ 0\ \tau_1)$
X-AssignOOB	$H; *n^{\tau} = n_1^{\tau_1}$	$\rightsquigarrow$	$H; \texttt{Bounds}$	where $\tau = \texttt{ptr}^c(\texttt{array}\ 0\ \tau_1)$
X-DerefNull	$H; *0^{\tau}$	$\rightsquigarrow$	$H; \texttt{Null}$	where $\tau = \texttt{ptr}^c\omega$
X-AssignNull	$H; *0^{\tau} = n_1^{\tau'}$	$\rightsquigarrow$	$H; \texttt{Null}$	where $\tau = \texttt{ptr}^c(\texttt{array}\ l\ \tau_1)$
X-AmperNull	$H; \&0^{\tau} \rightarrow f_i$	$\rightsquigarrow$	$H; \texttt{Null}$	where $\tau = \texttt{ptr}^c\texttt{struct}\ T$
X-BinopNull	$H; *0^{\tau} = n^{\tau'}$	$\rightsquigarrow$	$H; \texttt{Null}$	where $\tau = \texttt{ptr}^c(\texttt{array}\ l\ \tau_1)$

C-Exp

$$\frac{e = E[e_0] \qquad m = mode(E) \vee m = u \qquad H; e_0 \rightsquigarrow H'; e_0' \qquad e' = E[e_0']}{H; e \longrightarrow^m H'; e'}$$

C-Halt

$$\frac{e = E[e_0] \qquad m = mode(E) \vee m = u \qquad H; e_0 \rightsquigarrow H'; r \text{ where } r = \texttt{Null} \text{ or } r = \texttt{Bounds}}{H; e \longrightarrow^m H'; r}$$

$$\begin{array}{lcl}
mode(_) & = & c \\
mode(\texttt{unchecked}\ E) & = & u \\
mode(\texttt{let}\ x\ =\ E\ \texttt{in}\ e) & = & \\
\quad mode(E + e) & = & \\
\quad mode(n + E) & = & \\
\quad mode(\&E \rightarrow f) & = & \\
\quad mode((\tau)E) & = & \\
\quad mode(*E) & = & \\
\quad mode(*E = e) & = & \\
\quad mode(*n = E) & = & mode(E)
\end{array}$$

Figure 10. Semantics (complete)

B Graphed Results of Modifications and Overheads

A summary of the results of code changes is shown in Figure 11. A summary of the overheads these Checked C and these changes introduce is shown in Figure 12.

C Vsftpd porting experience

Vsftpd ("Very secure FTP daemon") is a modern FTP server designed with security in mind. After performing an initial port of vsftpd using our porting tool, we spent a few more days manually porting vsftpd. We were able to make a substantial amount of the code checked, and we estimate that quite a bit more could be made so. Here we describe some of our experience.

C.1 Checked strings

Vsftpd defines its own internal datastructure for managing strings that aims to avoid some of the well-known problems with zero termination (e.g,. due to use of functions like `strcpy`). Our ported version of this data structure is the following:

```
struct mystr
{
  _Nt_array_ptr<char> p_buf : count(alloc_b-1);
  unsigned int len; /* len <= alloc_bytes */
  unsigned int alloc_b;
};
```

The `p_buf` field contains the string data, totaling `alloc_b`. The code intends for `p_buf` to always be NUL terminated, but the NUL may (also) appear before the end of the buffer, at location `len`. When we ported this code to Checked C, we ended up ensuring location `s->p_buf[s->alloc_b-1]` also always contains a NUL. Here are some relevant prototypes of functions that operate on `struct mystr` objects:

```
_Nt_array_ptr<const char> str_getbuf(_Ptr<const
    struct mystr> p_str);
_Nt_array_ptr<const char> str_strdup(_Ptr<const
    struct mystr> p_str);
void
private_str_append_memchunk(_Ptr<struct mystr>
    p_str, _Array_ptr<const char> p_src : count
    (len), unsigned int len);
void str_empty(_Ptr<struct mystr> p_str);
```

The first function returns the `p_buf` field; the second makes a copy of `p_buf` (up to `len`) and returns that; the third appends a buffer to the existing string, starting at `len`, allocating more space if need be; the fourth empties the string (writing a NUL at position 0). This third function presented a problem for the current compiler: When reallocating to make more space, we need a way to atomically update `p_buf` and `alloc_b`, but at the moment this is not implemented by the compiler (so required an unchecked block).

C.2 System functions

Vsftpd makes heavy use of I/O libraries, managing sockets, network communication, file reading/writing, and more. It isolates the relevant functions in a single file, `sysutil.c`. Much of the code in this file can be made checked, but those bits that directly interact with low-level resources cannot be. E.g., calls to `mmap`, `read`, `write`, etc. require sending/receiving memory buffers, which we may cast to/from checked pointers when appropriate. In most cases, we could provide checked types for these functions. Here are some examples:

```
_Nt_array_ptr<char> vsf_sysutil_getcwd(
    _Array_ptr<char> p_dest : count(
    buf_size), const unsigned int buf_size)
     : count(buf_size);
int vsf_sysutil_read_loop(const int fd,
    _Array_ptr<void> p_buf : byte_count(
    size), unsigned int size);
_Nt_array_ptr<const char>
    vsf_sysutil_double_to_str(double
    the_double);
_Array_ptr<const unsigned char>
    vsf_sysutil_sockaddr_ipv6_v4(const _Ptr
    <struct vsf_sysutil_sockaddr>
    p_sockaddr) : count(4);
```

The first is a wrapper around a call to get the current working directory, sotring the result in `p_dest`. The second reads `size` bytes into `p_buf` from `fd`. The third coverts a double to a (statically allocated) string. The fourth converts a generic address object into a (4 byte) IPv4 address. For this last function we were able to provide a more precise type, since we could specifically indicate the size of the returned buffer.

C.3 Summary

In total, we ported 24 of the 39 files in vsftpd to be fully checked. This represents 6725 lines of code that are totally checked, compared to overall 16546 lines of code in the entire project. Additionally, checked types are used throughout the code, even in files which are not checked. Throughout the project, we re-wrote 2715 declarations of variables, parameters, and structure field members to use checked types.

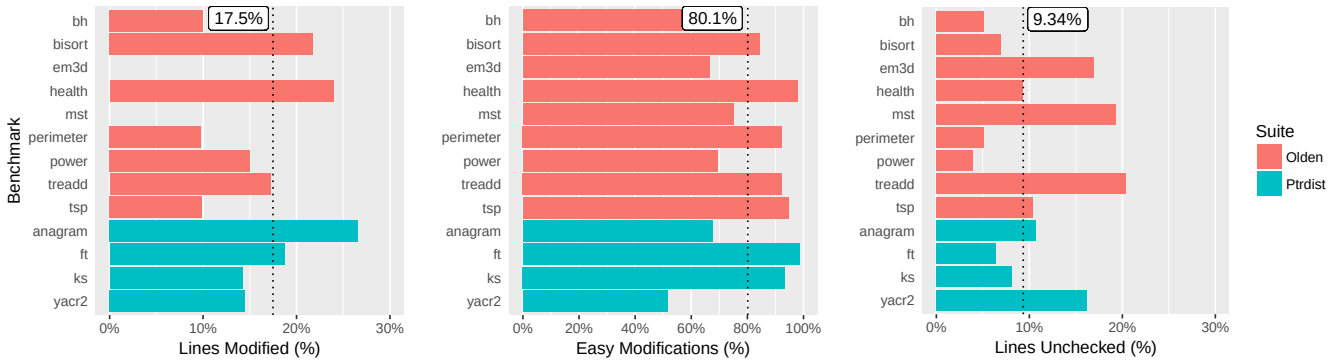

Figure 11. Summary of Code Changes

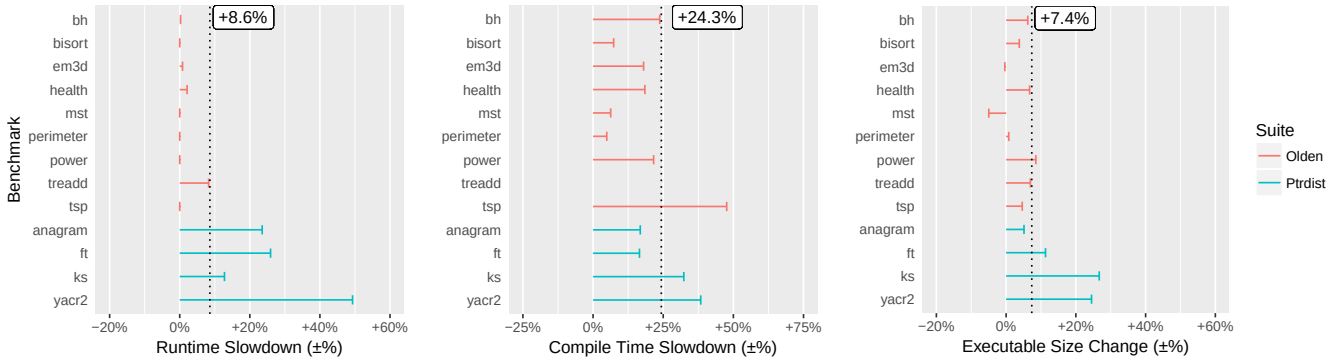

Figure 12. Summary of Modification Overheads