Putting the Checks into Checked C

Checked C Technical Report Number 2

31 October 2017

Archibald Samuel Elliott

Paul G. Allen School,
University of Washington

Summary

Checked C is an extension to C that aims to provide a route for programmers to upgrade their existing C programs to a safer language without losing the low-level control they enjoy. Checked C currently only addresses unsafe code with spatial memory violations such as buffer overruns and out-of-bounds memory accesses.

Checked C addresses these memory safety problems by adding new pointer and array types to C, and a method for annotating pointers with expressions denoting the bounds of the objects they reference in memory. To ensure memory safety, the Checked C compiler uses a mixture of static and dynamic checks over these additions to the C language.

This Technical Report concerns these Dynamic Checks, and the algorithms and infrastructure required to support them, including: the soundness property Checked C is aiming to preserve, propagation rules for bounds information, and the code generation algorithm for the checks themselves. This report includes an evaluation of these dynamic bounds checks, their overhead, and their interaction with a state-of-the-art optimizer.

Contents

1 Introduction **1**

1.1 Contributions 1

2 Overview of Checked C **3**

2.1 New Data Types 3

2.2 Bounds Expressions 4

2.3 Explicit Cast Operators 4

2.4 Checked Regions 4

2.5 Interoperation Types 5

2.6 Soundness 5

2.7 Explicit Dynamic Checks 6

2.8 Other Features 6

3 Example **7**

3.1 Unchecked Program 7

3.2 Conversion to Checked C 8

3.3 Compiler-inserted Dynamic Checks 8

3.4 Optimized Checks 9

3.5 Checked Function Calls 10

3.6 Corrected Checked C Program 12

4 Bounds Propagation **13**

4.1 Canonical Forms 13

4.2 C's Value Semantics 14

4.3 Propagation Rules 15

4.4 Narrowing 17

4.5 Propagation Limitations 17

5 Checks **19**

5.1 Design Requirements 19

5.2 Static Checks 19

5.3 Dynamic Checks 20

6 Preliminary Evaluation **24**
6.1 Benchmarks 24
6.2 Code Modifications 25
6.3 Performance Results 26
6.4 Evaluation 27

7 Related Work **28**

8 Conclusion **30**

Appendices

A Bibliography **31**

B End Notes **33**

C C Syntax **34**

D Propagation Rules **36**

Chapter 1

Introduction

The C programming language [14, 3] grants programmers low-level control of their programs rivalled only by directly writing assembly by hand. This is both its greatest strength, and its greatest weakness [6]. C grants control of memory safety (and many other kinds of safety) directly to the programmer, but unfortunately programmers are human and make mistakes, and these mistakes can be devastating.

We classify memory safety mistakes into one of two categories. *Temporal safety* errors happen when a programmer attempts to use an object outside of its defined storage duration. *Spatial safety* errors happen when a programmer attempts to use a pointer to an object to access memory beyond the extents of that particular object. A *buffer overrun* is a kind of spatial safety violation where a program reads memory surrounding an object in memory, usually because the programmer calculated its size incorrectly.

During the period 2012–2016, buffer overruns were the source of 9.7% to 18.4% of CVEs reported in the NIST vulnerability database [13]. During that time, buffer overruns were the leading single cause of CVEs.

Programmers are still unwilling to give up this low-level control of their programs, despite protections that type-safe languages may provide [5]. Fortunately we can build on prior research [4, 22] into language safety to devise a safer C which does not give up this low-level control, without the drawbacks of previous approaches.

The Checked C project is a new effort towards achieving a safer C. It does this by extending the language with new types and constructs for expressing invariants about objects in memory, and then enforcing these invariants within compiled programs using both static and dynamic checks.

Checked C currently only addresses spatial memory safety, and therefore only provides constructs for reasoning about the bounds of objects in memory. The language extension does not yet address any temporal memory safety guarantees. The Checked C specification is available online at `https://github.com/microsoft/checkedc/releases`.

This report describes my work on developing and implementing the dynamic checks for spatial memory safety and the infrastructure they require.

1.1 Contributions

I joined the project at the time the Checked C Specification v0.6 was released, in January 2017. At that point, the Checked C compiler could parse and represent its new types, and bounds expressions, but it did minimal verification of these bounds, and could not generate dynamic checks as required by the specification.

The Checked C Specification v0.6 [20] has the following to say about dynamic bounds checks:

> Given `*e1`, where `e1` is an expression of type `array_ptr<T>`, the compiler determines the bounds for `e1` following the rules in Section 4.2. Special rules are followed in `bundled` blocks to determine the bounds for `e1`. The compiler inserts checks before the memory pointed to by `e1` is read or written that `e1` is non-null and that the value of `e1` is in bounds.
>
> If `e1` ⊢ `bounds(e2, e3)`, the compiler inserts a runtime check that `e2 <= e1 && e1 < e3`. If the runtime check fails, the program will be terminated by the runtime system or in, systems that support it, a runtime exception will be raised. If `e1` ⊢ `count(e2)`, this is expanded to `e1` ⊢ `bounds(e1, e1 + e2)` before inserting checks. Of course a temporary variable would be used to hold the value of `e1`. — Extending C with bounds safety [20, Section 3.10]

While this describes the requirements of the dynamic bounds checks and the propagation algorithm that computes the bounds to check against, it is informal and vague.

My contributions to the Checked C project are:

- I was the first to clearly describe exactly where dynamic checks are required in terms of the C semantics (Section 5.3).
- I devised the flow-insensitive algorithm described in Chapter 4 which computes bounds required for checks using lvalues and values, which corresponds closely to the semantics of C expressions, values and lvalues.
- As part of devising this algorithm, I devised a coherent method for dealing with C structs and arrays, and their arbitrary composition, which keep checks sound and prevent some type errors, while also keeping run-time overhead low. This is called "narrowing", and is described in Section 4.4.
- I extended the existing clang code generator to generate the code to compute the required bounds and perform the required checks in LLVM IR (subsection 5.3.1).
- I proposed and performed the first evaluation of the overhead of Checked C's dynamic bounds checks. The main result is that, for the pointer-intensive benchmarks we chose, the run-time overhead is on average 8.6%, and at most 49.3%, with an average executable size overhead of 7.4% and at most 26.7%. Further results are described in Chapter 6.

This report also provides a quicker overview of Checked C than the specification (Chapter 2), and a worked example (Chapter 3) to illustrate how the compiler works in broad brushstrokes.

Chapter 2

Overview of Checked C

Checked C is an extension to C that adds static and dynamic checking to prevent and detect common programming errors such as buffer overruns, out-of-bounds memory accesses, and incorrect type casts. Given that it is an extension to C, it aims to preserve backwards compatibility with C11 [3].

This report is concerned with how Checked C prevents buffer overruns and out-of-bounds memory accesses. In order to reason about these behaviours, Checked C adds new data types, a method to annotate variable declarations with a static description of their runtime bounds, checked regions, new casts, and a method for interoperation with and describing existing C code.

It is this static bounds description, which we call a "bounds expression", that is used both during static analysis and during code generation for the dynamic checks, and that we consider to be the main idea underlying Checked C's approach.

2.1 New Data Types

Checked C adds new types to C:

- `ptr<T>`, the checked pointer to singleton type,
- `array_ptr<T>`, the checked pointer to array type,
- `T checked[N]`, the checked array type,

Both the checked pointer to singleton and the checked pointer to array types are intended to replace most uses of C's `T*`, which we call the unchecked pointer type. The checked array type is intended to replace most uses of C's `T[N]`, which we call the unchecked array type.

The checked pointer to singleton type, `ptr<T>`, is intended to replace uses unchecked pointer types that are a pointer to a single value of type `T`. For this reason, it is illegal to perform any pointer arithmetic on a value of type `ptr<T>`, including array indexing. Checked pointers to singletons require no bounds checks, but may be null.

The checked pointer to array type, `array_ptr<T>`, is intended to replace uses of the equivalent unchecked pointer type, where it is used as a pointer to an array of values of type `T`. These pointers may be null, may not overflow on pointer arithmetic, and require bounds checks when they are used to access memory.

The checked array type, `T checked[N]`, is intended to replace uses of the equivalent unchecked array type, `T[N]`. In the same way that there is a duality between C's unchecked arrays and unchecked pointers, there is the same duality between Checked C's checked pointers to arrays and checked arrays.

We place two restrictions on the checked array type, `T checked[N]`. First, they may not be variable length, so `N` must be a compile-time constant. Second, if an outer dimension of a multidimensional array is checked, all dimensions inside that dimension are also checked.

2.2 Bounds Expressions

In order to denote the bounds on a declaration or parameter, a bounds expression is associated with a checked array or checked array pointer, such as the bounds annotation at (A) in Listing 2.1.

```
int main(int argc, array_ptr<char*> argv : count(argc) (A));
```

Listing 2.1: Declaration of `main` in a Checked C program

Listing 2.1, though very short, shows several features of Checked C. The first is that Checked C's new types work exactly how C's regular types already do. The next feature is that we can annotate function parameters of pointer type with a bounds expression that provides a static description of that parameter's runtime bounds. The bounds expression at (A) is only for the outer `array_ptr<T>`, it does not apply to the inner unchecked pointer.

The last feature is the bounds expression itself. There are three forms of bounds expressions in Checked C. The first, called a "count expression" is shown above — here it denotes that the array has `argc` elements. There is a more complicated bounds expression called a "range expression", which is written `bounds(lower, upper)` and denotes that the associated pointer is within the range $[lower, upper)$. The last form of a bounds expression is written `byte_count(n)` and means the array pointer points to the start of an array n bytes long, regardless of the element size.

In Listing 2.1, we could not declare `argv` to have type `char* checked[argc]`, as `argv` checked arrays may not be variable-length. It is also not `char checked[argc][]`, both as this declares the outer array as variable-length, and as this requires the inner array to be checked, but we have no information as to its length as it is a null-terminated string.[1]

It is an error for a Checked C program to access memory via a null checked pointer, so these bounds only apply if the pointer is non-null.

2.3 Explicit Cast Operators

The logic of Checked C's bounds checks is reliant on C's expression semantics. Notably, this logic is undecidable, especially when it comes to deciding if one bounds expression implies another. We would prefer if compiling did not take infinitely long, so Checked C includes two operators that a programmer may use when the compiler requires, but cannot prove, that one bounds expression is a sub-range of another. These assert to the compiler that an expression has particular bounds, and are our "bounds cast" operators (in the same way a type cast operator asserts to the compiler that an expression has a particular type). Section 5.2 describes exactly when the compiler needs to prove these implications.

The dynamic bounds cast operator, `dynamic_bounds_cast<T>(e, lb, ub)`, performs a cast to type T and a dynamic check to make sure that either e is `NULL`, or that the required bounds, $[lb, ub)$, are a sub-range of the bounds of e. This check is explicit so that the programmer can see where there is an overhead in their program. Effectively this asserts a fact that will later be verified at run time.

The assume bounds cast operator, `assume_bounds_cast<T>(e, lb, ub)`, has the equivalent compile-time behaviour of the dynamic bounds cast operator, but performs no run-time check. This is used to assert to the compiler that e really has bounds $[lb, ub)$, without incurring run-time overhead. This asserts a fact that will never be verified, and is therefore a potential source of unsoundness.

2.4 Checked Regions

Checked C programs contain regions of unchecked and checked code. By default, all code is within an unchecked region. A programmer may annotate that a particular scope (i.e. a function or a block) is either a checked or unchecked region. A programmer may also use a pragma to say that any future scopes or declarations in the program file are part of a checked or unchecked region.

Within a checked region, the programmer may not use expressions or make declarations with unchecked types. In unchecked regions, a programmer may use checked types and they will still be checked against their bounds. Checked regions may call unchecked functions, as long as the latter's pointer parameter and return types are checked or have an interoperation type. Execution will also continue from a checked region into an unchecked region if a checked region contains an unchecked block (and vice versa).

2.5 Interoperation Types

Checked C includes a method to annotate regular C declarations with bounds or "interoperation" types. This is so that Checked C code can call unchecked functions and still make assumptions about how pointer parameters will be used, or returned pointers may be used. In checked regions, these declarations are treated as if they have their checked types and bounds.

These interoperation types are primarily used for upgrading code gradually. Checked C provides a copy of the C11 standard library headers files with each declaration annotated with its Checked C interoperation type, in order to allow programmers to use the standard library from inside checked programs.

2.6 Soundness

The main aim of the Checked C compiler is to maintain a moderately complex soundness condition over checked regions of the program. A Checked C program must use either static checks, dynamic checks, or a mixture of both to enforce this soundness condition.

Well-Formedness We define a Checked C checked region and its memory to be well-formed if:

- The values of all bounds expressions for in-lifetime variables or their transitively reachable data are defined and a sub-range of their (potentially overlapping) objects in memory; and,
- All in-lifetime non-null pointers of type `T` with bounds must point to an object in memory of type `T`.

Soundness A Checked C program is sound if, given a well-formed Checked C program and memory at the time when a checked region is entered:

- during the evaluation of an expression in that checked region, any newly computed or declared bounds are also well-formed with respect to the region and program memory;[2] and
- accesses (reads or writes) to memory through a pointer in the checked region, ensure that the checked region and program memory remain well-formed.

Corollary A corollary to this soundness is the fact that if a checked region completes evaluation, then we know that any reached memory was accurately described by its bounds information, and no checked pointers tried to access out-of-bounds memory. This implies a statement akin to the blame theorem from gradual typing [21], that, as long as all bounds-declarations are well-formed, any errors can be blamed on unchecked code.

Formalization This condition is further explored and formalized in our paper draft [16], where we provide a proof in the spirit of gradual typing, showing that (in a restricted core of the Checked C language) bounds errors in checked code can be entirely blamed on unchecked code.

Undefined Behaviour Checked C is very clear that operations which do not retain this well-formedness condition should raise an error, rather than causing "undefined behaviour", which is what C specifies in similar circumstances. This is a deliberate effort to ensure soundness of compiled programs and to reduce programmer confusion, at the cost of some run-time overhead.

2.7 Explicit Dynamic Checks

Checked C also provides an explicit dynamic check operator, `dynamic_check`. This works similarly to C's `assert`, taking a condition that must be true and signalling a run-time error if that condition evaluates to false. The main difference between the dynamic check built-in and assert is that dynamic check is never removed unless the compiler can prove the check will always pass, whereas an assert will be removed if a particular preprocessor macro is defined.

2.8 Other Features

Both the Checked C specification and our prototype compiler are currently incomplete. As far as the specification is concerned, we do not have a complete design for null-terminated arrays, which means we have no way of reasoning about C strings.

As far as the compiler is concerned, though we have a complete definition of how to deal with pointer alignment issues, our compiler has no support for these features at the moment. The specification also defines bundled blocks, which are for relaxing invariants inside a particular block (as long as the invariants are enforced again upon exiting that block). Bundled blocks are also currently unimplemented in our compiler.

I consider all these issues — null-terminated arrays, pointer alignment issues, and bundled blocks — to be out-of-scope of this report.

Chapter 3

Example

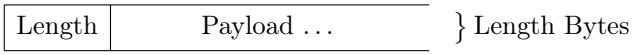

Figure 3.1: Echo Request Format

We are going to walk through converting a C program into Checked C. The program is part of a server that receives a request, in the format shown in Figure 3.1. It will then parse the message, and respond with a copy of the message, in the same format. The original C code will contain a vulnerability very similar to that in Heartbleed [18], which we will show that the Checked C version prevents.

3.1 Unchecked Program

The C function is shown in Listing 3.1. Our function, `echo`, is modelled as a response handler, which is provided with a complete request struct (D) containing data from the request parser, and an incomplete response struct (E) which the handler will construct the response data into.

This example has two vulnerabilities:

1. The first is based on Heartbleed [18], a notorious OpenSSL vulnerability disclosed in April 2014.

   In the request, the user is providing a payload (C) to be echoed back to them, and its length (A). The correct behaviour of the handler is to copy the data from the payload buffer into a buffer in the response (E).

   There is no requirement that the length provided in the request is correct, and if the programmer uses this user-provided length, as they have at (F), (G), and (I), then they may read beyond the bounds of the payload buffer at (J). If the programmer had used the length provided by their parser (B), then this buffer overflow could have been avoided.

   In the case of Heartbleed, this buffer overflow allowed attackers to view arbitrary segments of server memory.

2. The second vulnerability is a common error made by C programmers, that of not checking function return values. If the call to `malloc` at (F) returns `NULL`, then the allocation failed, and you may not use that pointer to access memory.

We have inlined the call to `memcpy` at (H) to show where a loop like this would dynamically fail. The behaviour is slightly different if we do not inline `memcpy`, and will be described later in Section 3.5.

```
   typedef struct req_t {  // Request Struct
Ⓐ    size_t length;         // user-provided
Ⓑ    size_t payload_len;    // parser-provided
Ⓒ    char  *payload;
     // ...
   } req_t;

   typedef struct resp_t {  // Response Struct
     size_t payload_len;
     char  *payload;
     // ...
   } resp_t;

   bool echo(req_t *req Ⓓ, resp_t *resp Ⓔ) {  // Handler
     char *resp_data   = malloc(req->length); Ⓕ

     resp->payload_len = req->length; Ⓖ
     resp->payload     = resp_data;

     // memcpy(resp->payload, req->payload, req->length); Ⓗ
     for (size_t i = 0; i < req->length Ⓘ; i++) {
       resp->payload[i] = req->payload[i] Ⓙ;
     }
     return true;
   }
```

Listing 3.1: Unchecked Example

3.2 Conversion to Checked C

In order to make this program safer, we will convert it into Checked C. This will protect us from both the vulnerabilities that it includes - the out-of-bounds array access, and accessing memory via the null pointer returned from `malloc`.

This conversion will proceed in two steps:

1. First, we will convert any declarations from their existing unchecked C types to their corresponding Checked C types, which relies on analysing how these declarations are used.

   In Listing 3.2, we start by converting the types at Ⓐ, Ⓑ, and Ⓔ from unchecked pointers into checked array pointers, as these are used as arrays. We also change Ⓒ and Ⓓ to checked singleton pointers, as we know these pointers are not used as arrays, they are directly accessed.

2. Second, we will annotate any array pointers with a description of their bounds in order to allow us to enforce these bounds at compile- or run-time.

   From the specification of the message format, we know that the payload is variable-length, the length field is not trustworthy, but we should relate the parser-provided length in the request (or the handler-provided length in the response) to the payload buffer. This is done with the bounds declarations at Ⓐ and Ⓑ, which reference the other members of the struct. We also annotate the local variable used as a temporary for the result of malloc, at Ⓔ, with its intended size.

3.3 Compiler-inserted Dynamic Checks

In order to preserve the soundness condition proposed in Section 2.6, the Checked C compiler must insert dynamic checks which ensure these invariants hold, if it cannot statically prove these invariants hold at

```
   typedef struct req_t {  // Request Struct
     size_t length;         // user-provided
     size_t payload_len;    // parser-provided
Ⓐ    array_ptr<char> payload : count(payload_len);
     // ...
   } req_t;

   typedef struct resp_t {  // Response Struct
     size_t payload_len;
Ⓑ    array_ptr<char> payload : count(payload_len);
     // ...
   } resp_t;

   bool echo(ptr<req_t> req Ⓒ, ptr<resp_t> resp Ⓓ) {  // Handler
Ⓔ    array_ptr<char> resp_data : count(req->length) = malloc(req->length);

     resp->payload_len = req->length;
     resp->payload     = resp_data;

     // memcpy(resp->payload, req->payload, req->length);
     for (size_t i = 0; i < req->length; i++) {
       resp->payload[i] = req->payload[i];
     }
     return true;
   }
```

Listing 3.2: Checked Example (Converted from Listing 3.1)

compile-time. To show how this works, Listing 3.3 is the same code as Listing 3.2, but with all implicit dynamic checks explicitly shown with the `dynamic_check` operator, and without the compiler attempting to prove any of them.

Note first the checks at Ⓐ and Ⓑ which ensure the checked singleton pointers are not `NULL`. We also check the checked array pointer at Ⓔ, which ensures we do not access memory using the pointer returned from `malloc` if it is `NULL`.

At Ⓒ and Ⓓ we check the bounds of the access to the request payload against its bounds. Here the bounds are computed using information from the same struct that the pointer is a member of. We check the invariants both for the read of the request payload, and (at Ⓕ and Ⓖ) for the write into the response payload.

Returning to our bugs, `malloc` returning `NULL` will be caught by Ⓔ, and attempting to read the request payload out-of-bounds will be caught by Ⓓ.

However, to claim that the program in Listing 3.3 is now bug-free would be incorrect, unless the programmer wishes for the program to crash on bad inputs.

3.4 Optimized Checks

Performing those checks in this inner loop is also not particularly practical, as these checks are slow. If a compiler can prove it is safe, they are free to move, optimize, or delete these dynamic checks if they can prove the program will remain sound. Listing 3.4 shows a version of Listing 3.3 where some checks have been hoisted and some have been removed.

In this example, the compiler has deleted four non-null checks, it has hoisted two loop-invariant non-null checks out of the inner loop, and it has deleted 3 bounds checks that can be proven to be unneeded using information about the range of `i` within the loop.

```
typedef struct req_t {  // Request Struct
  size_t length;         // user-provided
  size_t payload_len;    // parser-provided
  array_ptr<char> payload : count(payload_len);
  // ...
} req_t;

typedef struct resp_t {  // Response Struct
  size_t payload_len;
  array_ptr<char> payload : count(payload_len);
  // ...
} resp_t;

bool echo(ptr<req_t> req, ptr<resp_t> resp) {  // Handler
Ⓐ  dynamic_check(req != NULL);
  array_ptr<char> resp_data : count(req->length) = malloc(req->length);

Ⓑ  dynamic_check(resp != NULL);
  resp->payload_len = req->length;
  dynamic_check(resp != NULL);
  resp->payload     = resp_data;

  for (size_t i = 0; dynamic_check(req != NULL), i < req->length; i++) {
    dynamic_check(req != NULL);
    dynamic_check(req->payload != NULL);
Ⓒ   dynamic_check(req->payload <= &(req->payload[i]));
Ⓓ   dynamic_check(&(req->payload[i]) < (req->payload + req->payload_len));
    dynamic_check(resp != NULL);
Ⓔ   dynamic_check(resp->payload != NULL);
Ⓕ   dynamic_check(resp->payload <= &(resp->payload[i]));
Ⓖ   dynamic_check(&(resp->payload[i]) < (resp->payload + resp->payload_len));
    resp->payload[i] = req->payload[i];
  }
  return true;
}
```

Listing 3.3: Checked Example with Explicit Checks (Based on Listing 3.2)

The remaining check inside the loop has been simplified according to the semantics of C, and is exactly the check that fails if an attacker provides a length that is larger than their payload length. In the unchecked code, this is the condition that would cause a buffer overflow.

We currently make no guarantees that we are able to perform any of these check elisions or optimizations, but it is our aim to be able to eventually.

3.5 Checked Function Calls

Had the programmer not written their own copy loop, and instead used `memcpy`, their results would have overall been the same, but static analysis would have told them slightly sooner about their error.

In Checked C we ship a copy of the C standard library declarations, annotated with interoperation types so that they may be used in Checked contexts. The declaration of `memcpy` used in checked scopes is shown in Listing 3.5.

In this case, the compiler will not insert dynamic checks if the programmer calls `memcpy`, as has been commented out in Listing 3.2, because none of the checked pointer arguments are being used to access memory at the

```
typedef struct req_t {  // Request Struct
  size_t length;         // user-provided
  size_t payload_len;    // parser-provided
  array_ptr<char> payload : count(payload_len);
  // ...
} req_t;

typedef struct resp_t {  // Response Struct
  size_t payload_len;
  array_ptr<char> payload : count(payload_len);
  // ...
} resp_t;

bool echo(ptr<req_t> req, ptr<resp_t> resp) {  // Handler
  dynamic_check(req != NULL);
  array_ptr<char> resp_data : count(req->length) = malloc(req->length);

  dynamic_check(resp != NULL);
  resp->payload_len = req->length;
  resp->payload     = resp_data;

Ⓐ dynamic_check(req->payload != NULL);
Ⓑ dynamic_check(resp->payload != NULL);
  for (size_t i = 0; i < req->length; i++) {
Ⓒ   dynamic_check(i < req->payload_len);
    resp->payload[i] = req->payload[i];
  }
  return true;
}
```

Listing 3.4: Optimised Checked Example with Explicit Checks (Based on Listing 3.3)

```
void *memcpy(void * restrict dest : byte_count(n),
             const void * restrict src : byte_count(n),
             size_t n)
             : bounds(dest, dest + n);
```

Listing 3.5: Checked Type of `memcpy`

call (the non-null checks of `resp` and `req` will remain as these are accessed to find the value of the pointers in the call expression).

Instead the compiler must statically prove that the bounds calculated for the `resp->payload` argument are at least as wide as the declared bounds of the `dest` parameter, and also for `req->payload` and `src`. In this case, both argument values are of `char*` type, so `byte_count(x)` expressions are equivalent to `count(x)` expressions.

Using congruence, the compiler should be able to prove this property for `dest`, but it will be unable to for `src`, as it has no information to relate `req->payload_len` and `req->length`. This will cause a compile-time error.

This example is exactly what the dynamic cast operators from Section 2.3 are for. Providing the argument `dynamic_cast<array_ptr<char>>(req->payload, req->payload, req->payload + req->length)` for the `dest` parameter, which tells the compiler that the first argument has bounds [`req→payload`, `req→payload` + `req→length`), would satisfy the static check (preventing the compiler error), but this would also insert a dynamic check before the call to make sure that these bounds are larger than the original bounds of `req->payload`. If a user submits a packet where the length field is larger than the payload length, then this check would fail before the call to `memcpy`.

```
typedef struct req_t {  // Request Struct
  size_t length;         // user-provided
  size_t payload_len;    // parser-provided
  array_ptr<char> payload : count(payload_len);
  // ...
} req_t;

typedef struct resp_t {  // Response Struct
  size_t payload_len;
  array_ptr<char> payload : count(payload_len);
  // ...
} resp_t;

bool echo(ptr<req_t> req, ptr<resp_t> resp) {  // Handler
(A) if (req->payload_len < req->length)
      return false;

    array_ptr<char> resp_data : count(req->payload_len (B)) = malloc(req->payload_len (C));
(D) if (resp_data == NULL)
      return false;

    resp->payload_len = req->payload_len; (E)
    resp->payload     = resp_data;

    for (size_t i = 0; i < req->payload_len (F); i++) {
      resp->payload[i] = req->payload[i];
    }
    return true;
}
```

Listing 3.6: Corrected Checked Example (Based on Listing 3.2)

3.6 Corrected Checked C Program

Returning to the notion of making sure this program is not-only memory safe but also bug-free, Listing 3.6 shows the corrected program, where we have: validated the relationship of `req->length` and `req->payload_len` at (A), validated the return value of `malloc` at (D), and replaced all further used of `req->length` with `req->payload_len` at (B), (C), (E), and (F).

Chapter 4

Bounds Propagation

We have covered bounds expressions on declarations in Chapter 2. The other side of this coin is that we have to perform propagation to calculate the bounds on pointers at access time. As pointer bounds expressions are currently flow-insensitive, this can be done with a set of context-free rules that I will sketch here, and show fully in Appendix D.

Our bounds propagation algorithm works in two steps. The first step normalizes existing bounds expressions into their most general form, and then the main propagation rules work over these canonical forms to compute the bounds we should check any pointer access with.

4.1 Canonical Forms

We have been careful to define our bounds expressions such that they can be normalized into a single form, as shown in Table 4.1. This normalization makes bounds propagation easier, rather than having to ensure propagation works on every kind of bounds expression. This should also allow us to add new kinds of bounds expressions without having to change the propagation algorithm.

Checked array pointers with an explicit bounds expression are not the only kind of expressions we can compute bounds for. With this algorithm we may also need to compute the bounds of checked singleton pointers, and of the pointers derived from checked arrays (exactly when an array turns into a pointer we will cover shortly).

Bounds Kind	Declaration	Canonical Bounds
Range	`array_ptr<T> p : bounds(l, u)`	`bounds(l, u)`
Count	`array_ptr<T> p : count(n)`	`bounds(p, p + n)`
Byte Count	`array_ptr<T> p : byte_count(n)`	`bounds(p, ((array_ptr<char>)p) + n)`
Singleton	`ptr<T> p`	`bounds(p, p + 1)`
Array	`T a checked[N]`	`bounds(a, a + N)`

Table 4.1: Canonical Bounds Expressions. In Canonical Bounds, the + refers to C's pointer-integer addition operator, which adds integer multiples of the size of the pointer's referent type to the original pointer (hence the cast to `array_ptr<char>` in the `byte_count` case). The bounds on *Array* are used when `a` is converted from an array into a pointer, and `N` must be constant.

4.2 C's Value Semantics

Before we can accurately cover our propagation rules, we need to cover a discussion of how C's lvalues and values work[3].

In C, there are two kinds of values. Lvalues describe the locations of objects in memory, and are used to both read from and write to memory. Integers, floats, and pointers are values.

Every expression in C evaluates to either an lvalue or a value. We will call an expression that evaluates to an lvalue an *lvalue expression*; and likewise an expression that evaluates to a value a *value expression*. Usually expressions in C require their sub-expressions to be value expressions. Some expressions require certain of their sub-expressions to be lvalue expressions. Figure 4.1 shows all lvalue expression types and all places they may be used in C — all other expressions are value expressions and have only value sub-expressions.

As our invariants are all about how programs access memory, and programs can only access memory using an lvalue expression, our bounds propagation rules are inextricably tied to the notion of lvalue expressions and value expressions.

LValue Expressions	$e_l ::= x$	Variables
	$\mid *e_v$	Dereference
	$\mid e_v[e_v]$	Index
	$\mid e_l.m$	Struct Member Access
	$\mid e_v \rightarrow m$	Pointer Member Access
Value Expressions	$e_v ::= \&e_l$	Address-of
	$\mid e_l = e_v$	Assignment
	$\mid e_l \oplus= e_v$	Compound Assignment
	$\mid e_l{+}{+} \mid {+}{+}e_l$	Increment
	$\mid e_l{-}{-} \mid {-}{-}e_l$	Decrement
	$\mid e_v.m$	Struct Member Access
	$\mid \ldots$	Other Value Expressions
	$\mid e_l$	Lvalue & Array Conversion

Figure 4.1: Lvalue expressions and their usage in C. A more complete description of C's expression syntax, including a definition of $\oplus$ (binary operators), is contained in Appendix C. Note that a Struct Member Access can be an lvalue or a value depending on whether the sub-expression is an lvalue or a value (respectively).

The subtle nuance in lvalue expression evaluation is that lvalues may also appear in a value position in an expression. In this case, one of two things happens: if the lvalue has array type, the lvalue expression evaluates to a pointer to the start of that array. If the lvalue does not have array type, then the lvalue is read to find the underlying value stored at its location. These operations are respectively called an *array conversion*, and an *lvalue conversion*.

Thus C differentiates implicitly between lvalues, which denote locations, and the objects in those locations. The result of performing this lvalue conversion is the value in the location denoted by that lvalue, which we refer to as an *lvalue target*.

4.3 Propagation Rules

Our bounds propagation rules, at a broad level, will work with these three different evaluation models, separately (but mutually recursively) propagating bounds for value expressions, *value bounds*, for lvalue expressions, *lvalue bounds*, and for the value result of an lvalue conversion, *lvalue target bounds*. The full rules are shown in Appendix D.

In general, the value bounds of an expression are the value bounds of the pointer sub-expression with bounds; the lvalue bounds of an expression correspond to the bounds on the storage location of that lvalue; and the lvalue target bounds of an expression correspond to the bounds declared for that lvalue.

During propagation we add two more kinds of bounds annotation: bounds(any), which is equivalent to an infinite range, and represents the bounds on `NULL`; and bounds(unknown), which is equivalent to an empty range, and represents that we cannot calculate a static description of the run-time bounds.

4.3.1 Lvalue Bounds

The propagation rules for lvalue bounds of an lvalue expression are shown fully in Figure D.1. The intuition behind the bounds these rules calculate is that they are the bounds on the memory used for the lvalue itself, for instance on the stack. The bounds declarations are not used in lvalue bounds propagation.

Variables A (non-array-typed) variable only needs enough space to store one value of the type it holds, so pointer-typed variables only uses enough space for one pointer. This is what *single-bounds*(e_v) computes.

In the case of array-typed variables, the lvalue itself is an array, so the amount of space used can be deduced from the array size and type, if the array is constant-sized. This is what *array-bounds*(e_v, N) computes. If the array is not constant-sized, then we cannot deduce the bounds statically.

Dereferencing and Indexing In the case of pointer dereferencing or indexing, the location being accessed is the memory that the pointer references, which is exactly what the value bounds on that pointer describe. In the case of an indexing expression, $e_v[e_v]$, C makes no requirements on which sub-expression is the pointer, so we define the notion of the *base* of an expression, which in this case refers to the pointer-typed sub-expression.

Member Accesses In the case of struct and pointer memory accesses, the lvalue bounds we propagate match the location of the struct member in memory, including if it is an array or not. A further description of this behaviour is included in Section 4.4.

4.3.2 Value Bounds

The propagation rules for value bounds of expressions are shown in Figure D.2. The intuition behind the bounds these rules calculate is that these bounds expressions effectively propagate the lvalue target bounds up from any lvalue conversions.

Null Pointers The null pointer, `NULL`, always has bounds(any), as we allow any pointer with bounds also to be `NULL`. We will check the pointer is non-null before dereferencing.

Literals All other literals have unknown bounds, as they are not pointer literals.

Address-of Operator The address-of operator converts an lvalue sub-expression into a pointer representing its memory location. Thus the bounds on this pointer must be the lvalue bounds of the lvalue.

Assignment In C, assignment and compound assignment are expressions, not statements, so can appear within other expressions. The resultant value is the same as the value that is assigned into the lvalue, so in the case of assignment the bounds are the bounds of the right hand side, and in the case of compound assignment the bounds are the lvalue target bounds of the left hand side. Increment and decrement are just special cases of compound assignment.

Struct Member In the case of struct member expressions, we narrow to the field of the struct as we do when the base expression is an lvalue expression. Our implementation does not yet cope with this, as we have no way of referring to the base value expression from which we want to calculate the bounds.

Function Calls For function declarations, we allow the programmer to specify parameter and return bounds in terms of other parameters. Therefore, to calculate the bounds on a value returned from a function call, we need to substitute any argument values into the bounds expression of the return value.

Comma Expressions As comma expressions evaluate to their final expression, we propagate the value bounds of the final expression.

Array and Lvalue Conversions In the case of non-array-typed lvalues, because we will be reading the lvalue in order for it to become a value, the bounds on this conversion expression refer to the bounds on the object the lvalue denotes, i.e. the lvalue target bounds of that lvalue expression. In the case of array-typed lvalues, evaluation will perform an array conversion, converting the array into a pointer to its first element, which corresponds to the lvalue bounds on that array.

Binary Operators Binary operator expressions returning a pointer type, such as performing pointer arithmetic, take the bounds of the base expression, i.e. the sub-expression that has pointer type. This ensures that for a pointer, p, $p + 3$ and $p - 2$ have the same bounds as p does. The notion of "base" here is the same as with array indexing.

Relational and Logical Operators Relational operators return a boolean result, i.e. an integer 0 or 1 corresponding to the comparison result. These do not denote anywhere in memory, so do not have bounds. The same applies to the unary negation operator.

Other Expressions For other expressions, including arithmetic over integers (but not floats), we propagate bounds from sub-expressions to expressions. This allows us not to lose bounds information when performing bit-wise operations over pointer values (for pointer alignment, for example). The requirement we make is that if more than one sub-expression has bounds, we can only merge bounds if they are syntactically equal. We may later relax this to being syntactically equal modulo variable equality. In the case of float expressions, we do not believe performing pointer arithmetic with floating-point numbers makes any sense, so float expressions may not carry or propagate bounds information.

4.3.3 Lvalue Target Bounds

The propagation rules for value bounds of expressions are shown in Figure D.3. The intuition behind the bounds these rules calculate is that these bounds represent the extent in memory of the value returned when an lvalue conversion is performed.

Checked Singleton Pointers The bounds of a `ptr<T>`, `p`, start at their value, and include exactly enough space for a single value of their own type, `T`. They are $[\texttt{p}, \texttt{p} + \texttt{sizeof(T)})$.

Variables The lvalue target bounds for a variable declared with bounds is exactly the bounds expression it was declared with.

Member Accesses Struct and pointer member accesses both use the declared bounds on the member to compute their bounds expressions. In the case of struct member accesses, they use the value in the same struct of any other members referenced in the bounds expression. In the case of pointer member accesses, any members referenced in the bounds expression are also computed using a pointer member access of the same pointer.

For instance, given a struct, *s*, with two members, *m1* and *m2*, where *m2* has declared bounds of `count(m1)`, the lvalue target bounds of *s.m2* are $[s.m2, s.m2 + s.m1)$. If we had a pointer, *p*, to the same type of struct, the lvalue target bounds of $p{\to}m2$ would be $[p{\to}m2, p{\to}m2 + p{\to}m1)$. In the same way, any variables referenced in expressions on struct members are assumed to refer to other members of the same struct.

Dereferencing and Indexing We cannot propagate lvalue target bounds for a pointer that is stored at another pointer (unlike multidimensional arrays). This is because we have no way of expressing where the bounds of the inner pointer are stored or should be computed, unlike in a struct where there is potentially storage space.

4.4 Narrowing

One important part of our approach is how we choose to "narrow" bounds when accessing members of structs or elements of arrays. This means potentially inferring a sub-range of the aggregate object's bounds for accesses to inner parts of that object.

It is this narrowing that ensures the second part of the well-formedness condition in Section 2.6, that all in-lifetime non-null pointers of type *T* with bounds must point to an object in memory of type *T*.

In the case of structs (and unions), we have to narrow to the bounds sub-range for a particular member, because different members have different types. This is what the functions like *struct-rel-bounds*(e_v, m) do, narrowing the bounds inferred to the bounds of the field m within e_v.

In the case of arrays, our semantics allow us two implementations. The first is that we do narrow through each dimension of a multidimensional array, to ensure an access, `a[i][j]`, to a 5 by 5 array, is within the 5 elements starting at `a[i][0]`. Our compiler actually chooses the other alternative, ensuring that any access to `a[x][y]` is within the 25 elements starting at `a[0][0]`. This maintains well-formedness as arrays all contain elements of the same type, and should allow the compiler to remove some bounds checks from tight inner loops.

4.5 Propagation Limitations

We have several limitations with the current propagation algorithm, the most major of which relates to "modifying expressions", as defined in [20, Section 3.4], which pose a problem to the flow-insensitive algorithm. In particular, our algorithm does not introduce temporaries for expressions which are modifying, instead duplicating these expressions into the bounds expressions, and then computing based off the duplicates. This means if we generate directly from the bounds expressions, each modifying expression could be generated in code several times.

For instance, say I have an array, a of structs, each with a member $m1$ that has bounds defined to be `count(m2)`, where $m2$ is another member of the same struct. In this case, an access like `a[i++].m1[0]` will have the bounds inferred of `bounds(a[i++].m1, a[i++].m1 + a[i++].m2)`. If we generate dynamic checks directly from bounds expressions, we have now incremented `i` four times (once for the pointer itself, once for the lower bound, twice for the upper bound), which changes the meaning of the program.

This also is a problem if the programmer calls a function that returns a struct, and directly (without assigning into a temporary) accesses a member of the returned struct. We have no way of referring to the place in memory of the returned struct.

In both of these cases, the programmer can manually insert a temporary variable to avoid these limitations.

We are planning to solve this issue with a set of dynamically-scoped keywords that will refer to the current expression value, the current struct base pointer, and the current return value, which we would then treat as temporaries during code generation. While it makes the code generation more complex, it means we can infer the bounds in more places.

We also totally lack a way of handling ternary expressions during bounds propagation. Arguably we can generate new bounds expressions with ternary expressions in both the upper and the lower expressions, but this will make any static analysis much harder in the long term. We may introduce a new form of bounds expression to handle these cases explicitly.

Chapter 5

Checks

This section describes how we have implemented our prototype Checked C compiler. Our compiler, based on the Clang/LLVM toolchain, is available online at `https://github.com/Microsoft/checkedc-clang`.

5.1 Design Requirements

The Checked C project has certain aims for the language which influence the design of its checks, for instance where we rely on static checks and where we rely on dynamic checks.

The main aim of Checked C is that we can upgrade code incrementally from C to Checked C, which means, among other things, that we must be layout compatible with existing C code. We may not, for instance, change the pointer representation, as this would break the data layout of existing code and change the calling convention.

A second aim is that we should keep any overhead of dynamic checks only to where the programmer accesses memory and they are required for soundness. This means that any overhead should be understandable and predictable. On the other hand we will have to perform checks statically at compile-time on pointer assignments and function calls to ensure soundness.

A final aim is that Checked C should be able to be used wherever C can currently be used, including embedded devices. This means we do not require that programs are linked to runtime libraries which implement the checks, so that executable size remains small.

5.2 Static Checks

At assignments into variables with bounds, Checked C requires that the bounds of the value being assigned into the variable imply the bounds of the variable. In particular, this means that the bounds of a variable are a (non-strict) sub-range of the bounds of the expression being assigned into it. This ensures that any future memory accesses via the newly-assigned variable cannot access memory outside the bounds of the original expression.

C's semantics for function calls behave as if the programmer has assigned the argument expressions to the parameter declarations, and so are covered by this same static checking, only without any implied mutual ordering, and bounds on function parameters can reference other declared parameters.

We also require checks on updates to variables and members involved in bounds checks, such that they continue to denote the bounds of these objects in memory. For example, increasing the value in a length member of a struct also containing an pointer to an array of that length, without a corresponding `realloc` or other knowledge that the bounds of the array are actually longer, could lead to an out-of-bounds memory access, even if this operation is performed within a checked region.

Lastly, a static check is required when casting an expression to a ptr<*T*>type. This check ensures that the value being cast is within the implied bounds of the ptr<*T*>type so that this checked singleton pointer can be dereferenced without needing to check its range.

In all these cases, where the compiler cannot prove these facts, it may use the explicit dynamically-checked cast operators described in Section 2.3 to prove these facts dynamically rather than statically.

Further discussion of these static checks, and how exactly we "prove" anything about C programs to the compiler will be described in a future Checked C Technical Report.

5.3 Dynamic Checks

As mentioned previously, we are only interested in inserting dynamic checks in expressions that actually access memory. We can also only insert dynamic checks where we have bounds expressions for the pointer we're accessing memory through.

As described in Section 4.2, all C memory reads are via lvalue conversions, and all writes are via assignment or increment operators. This means we can add the bounds checks during the evaluation of the lvalue sub-expressions of these operators, if the accesses dereference checked pointers. This sidesteps generating dynamic checks on the lvalue sub-expression of the C & operator, which does not access memory (it only computes a pointer value).

In particular, we will add dynamic checks into the evaluation of the lvalue expression in lvalue conversions; on the left of the assignment and compound assignment operators; and on the evaluation of the lvalue expression in increment and decrement operators. We also add dynamic checks to the base expression in a struct member access, if the base expression is an lvalue, to ensure each level of access into the struct is within bounds, as required by our narrowing rules in Section 4.4.

We currently use a lazy propagation algorithm which will only propagate bounds information from declarations to uses which require the bounds for either static or dynamic checks. This avoids computing bounds for unchecked memory accesses.

5.3.1 Generating Dynamic Checks

The Clang/LLVM toolchain consists of a C compiler (Clang itself), a general-purpose optimizer and code generator, and various other compiler tools. The central part of the compiler is LLVM's Intermediate Representation (LLVM IR), a typed, high-level assembly language which abstracts away many machine and compilation details while still providing a language that is at an abstraction level closer to assembly languages.

The architecture of the Clang/LLVM toolchain is fairly conventional. Our fork of Clang parses Checked C, performs Checked C-specific static checks, and then generates unoptimized LLVM IR. LLVM then analyses and optimizes LLVM IR, before generating platform-specific assembly. Our fork of the Clang/LLVM toolchain only includes changes to Clang, we have not needed to change LLVM.

Clang's LLVM IR Generator is actually made up of five mutually recursive code generators which process the Checked C syntax tree. In terms of the C semantics above, four of these code generators generate code for value expressions—namely scalar values (such as integers, characters and floats), aggregate values (such as arrays and structs), vector values (for vectorized code), and complex number values. The code generator we are interested in, however, is the fifth, which is responsible for generating the LLVM IR for lvalue expressions.

In C, we already know the uses of C lvalues that access memory (described in Section 5.3), so we extend the code generation of LLVM for these lvalues to insert the required dynamic checks. There are three kinds of checks we currently insert, non-null checks (required for any checked pointer); bounds checks (required only for lvalues that access memory through a checked array pointer); and dynamic cast checks (required for casts from Section 2.3). If a pointer requires both a non-null check and a bounds check, the non-null check is performed first.

LLVM IR The following is a quick introduction to LLVM IR [7]. LLVM IR is typed, with instructions containing explicit argument type annotations. The two types we care about here are LLVM's pointer type, denoted `T*`, and LLVM's boolean type, denoted `i1`. Both Checked C's checked pointers, and C's unchecked pointers translate to LLVM pointers. LLVM pointers are a different type to LLVM's integers, but can be compared using the same integer comparison operators.

LLVM's integer comparison instruction, `icmp` is parameterized by the kind of comparison, here we use `eq` (equal), `ne` (not equal), `ult` (unsigned less than), and `ule` (unsigned less than or equal to). Unlike in C, LLVM's integer comparisons are fully defined for pointers, even if the pointers point to separate objects.

LLVM has a high-level `call` instruction for calling other functions and intrinsics. LLVM's conditional branch instruction, `br` always takes a boolean condition, and two labels, the first for if the condition is true, the second for when it is false.

Non-null checks As shown in Listing 5.1, these have fairly conventional generated code, involving comparing the computed pointer to `null` (the LLVM keyword for the null pointer), branching on the result. Importantly this branch has to happen before we access memory using the computed value.

One optimization we perform with non-null checks is for pointer member expressions, $e_v \rightarrow m$. We could check that the pointer computed for the member, m, is non-null, but this computed pointer will be different for each member in the struct, meaning different non-null checks per-pointer, that will potentially not be optimized away. Instead, we do a non-null check on the struct base, e_v, which increases check redundancy, allowing more optimization, without affecting soundness.[4]

```
  ; ... function prefix
  %ptr = ; ... computes pointer value
  %ptr_non_null = icmp ne <ty>* %ptr, null
  br i1 %ptr.non_null label %success, label %fail

%success:
  ; ... function continues
  ; uses of %ptr are valid

%fail:
  call void @llvm.trap()
  unreachable
```

Listing 5.1: Example Non-null check generated by the Checked C compiler.

Bounds Checks As shown in Listing 5.2, these are very similar to the non-null checks. After we have computed the pointer value (which has usually happened before a non-null check not shown in the example), we compute LLVM values for the upper bound and lower bound directly from the parts of the bounds expression as propagated by the algorithm in Section 4.3. Importantly, the pointer can be equal to the lower bound, but it has to be strictly less than the upper bound, to agree with the invariants in Section 4.1.

Dynamic Cast Checks In this case the checks are a good deal more complicated. If the value being cast is `NULL`, then there is no need to check the bounds, as the null pointer is within any bounds. Otherwise, the code ensures that the cast bounds are a sub-range of the bounds of the pointer. If either check succeeds, the pointer is cast to the new LLVM type, and execution continues.

Trap Blocks In all these checks we generate "trap blocks", which are the LLVM Basic Blocks that generate and signal the run-time exception for when the check fails. These consist of two instructions, a call to the LLVM trap intrinsic, `@llvm.trap()`, which the LLVM code generator knows how to turn into platform-specific

```
  ; ... function prefix
  %ptr = ; ... computes pointer value
  %ptr_lb = ; ... computes pointer lower bound
  %ptr_ub = ; ... computes pointer upper bound
  %ptr_in_lb = icmp ule <ty>* %ptr_lb, %ptr ; lower check
  %ptr_in_ub = icmp ult <ty>* %ptr, %ptr_ub ; upper check
  %ptr_in_bounds = and i1 %ptr_in_lb, %ptr_in_ub
  br i1 %ptr_in_bounds label %success, label %fail

%success:
  ; ... function continues
  ; loads and stores to %ptr are valid

%fail:
  call void @llvm.trap()
  unreachable
```

Listing 5.2: Example Bounds check generated by the Checked C compiler.

trap instructions; and an `unreachable` instruction so that the LLVM static analyser knows that execution does not continue after this branch.

We insert trap blocks at the end of the function so that the successful control flow is emphasized over the failure control flow. We also insert one of these per dynamic check (rather than all the dynamic checks sharing a single trap block per-function) so that debugging when optimizations are disabled is far easier. The Clang code generator automatically takes care of generating debug information for our generated code.

Gotchas As we use the scalar value code generator to generate the values for the upper and lower bounds checks, we have to be very careful that bounds checks are not recursive. A program is allowed to dereference checked pointers to compute bounds, however it is not allowed to dereference itself to compute its own bounds—doing so would lead to an infinite loop in check generation. We can almost entirely prevent this by only allowing bounds expressions to reference prior declarations, but care is needed around potential pointer aliasing.

5.3.2 Operating System Support For Dynamic Checks

We include no OS-specific behaviour for our dynamic checks, despite the fact that modern operating systems provide coarse-grained fault mechanisms for some amount of memory safety.

For instance, Linux, Mac OS X, and Windows will all signal a run-time error if a program attempts to access unmapped memory, including the first page at the `NULL` address. Our Checked C compiler could rely on this mechanism for performing some of the above dynamic checks, but does not.

Our compiler avoids these mechanisms for two reasons. The first is that we want to be able to deploy Checked C anywhere, including platforms without a full operating system to perform this check on our behalf. Secondly, the POSIX standard includes a mechanism for allowing the program to override the handling behaviour of or to ignore this run-time error. Doing so could affect program soundness.

We definitely cannot use this mechanism to implement bounds checks, as the ranges of mapped memory are too coarse-grained for our propagation and narrowing algorithm.

```
  ; ... function prefix
  %ptr = ; ... computes pointer value
  %ptr_null = icmp eq <ty>* %ptr, null
  br i1 %ptr_null, label %success, label %subrange_check

%subrange_check:
  %ptr_lb = ; ... computes pointer lower bound
  %ptr_ub = ; ... computes pointer upper bound
  %cast_lb = ; ... computes cast lower bound
  %cast_ub = ; ... computes cast upper bound
  %cast_in_lb = icmp lte <ty>* %ptr_lb, %cast_lb ; lower check
  %cast_in_ub = icmp lte <ty>* %cast_ub, %ptr_ub ; upper check
  %cast_subrange = and i1 %cast_in_lb, %cast_in_ub
  br i1 %cast_subrange, label %success, label %fail

%success:
  %cast_ptr = bitcast <ty>* %ptr to <ty>*
  ; ... function continues
  ; loads and stores to %cast_ptr are valid

%fail:
  call void @llvm.trap()
  unreachable
```

Listing 5.3: Example Bounds Cast Check generated by the Checked C compiler.

Chapter 6

Preliminary Evaluation

I converted two existing C benchmark suites as an initial evaluation of the consequences of porting code to Checked C (with assistance from David Tarditi and researchers at Samsung Research). We quantify both the changes required for the code to become checked, and the overhead imposed on compilation, running time, and executable size.

6.1 Benchmarks

Name	LoC	Description
bh	1,162	Barnes & Hut N-body force computation algorithm
bisort	262	Sorts using two disjoint bitonic sequences
em3d	476	Simulates electromagnetic waves in 3D
health	338	Simulates Colombian health-care system
mst	325	Computes minimum spanning tree using linked lists
perimeter	399	Computes perimeter of a set of quad-tree encoded images
power	452	The Power System Optimization problem
treadd	180	Computes the sum of values in a tree
tsp	415	Estimates solution for the Traveling-salesman problem
voronoi	814	Computes 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	Computes minimum spanning tree using Fibonacci heaps
ks	549	Schweikert-Kernighan graph partitioning
yacr2	2,529	VLSI channel router

Table 6.1: Compiler Benchmarks. Top group is the Olden suite, bottom group is the Ptrdist suite. LoC includes all comments and blank lines in benchmark source files. Descriptions are from [15, 1]. We were unable to convert *voronoi* from the Olden suite and *bc* from the Ptrdist suite using the current version of Checked C.

I chose the Olden [15] and Ptrdist [1] benchmark suites, described in Table 6.1, because they are specifically designed to test pointer-intensive applications, and they are the same benchmarks used to evaluate both Deputy [22] and CCured [12].

We evaluate Checked C using these benchmarks in two ways. First, we quantify the number and type of source code changes required to convert these benchmarks from C to Checked C. Second, we quantify the overhead of the run-time checks on benchmark run time, compile time, and executable size. The evaluation results are presented in Table 6.2.

Experimental Setup 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.

Excluded Benchmarks We were unable to convert the *voronoi* benchmark from the Olden suite due to bounds propagation limitations detailed in Section 4.5. We were unable to convert the *bc* from the Ptrdist suite due to lack of time. They are excluded from any conversion results.

	Code Changes			Observed Overheads		
Benchmark	*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
Geo. Mean:	17.5	80.1	9.3	+ 8.6	+ 24.3	+ 7.4

Table 6.2: Benchmark Results. Key: *LM %*: Percentage of Source LoC Modified, including Additions; *EM %*: Percentage of Code Modifications deemed to be Easy (see 6.2); *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)

6.2 Code Modifications

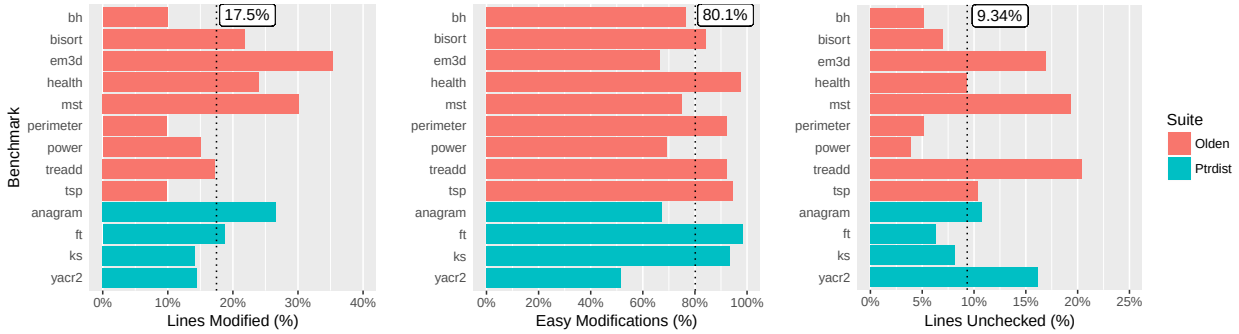

Figure 6.1: Code Modifications

On average, the changes modified 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 [2], 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, or assignments where our static analysis could not currently verify that the assignment was valid, 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 causes were almost entirely variable argument functions such as `printf`.

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.

We distinguish between the two because we believe easy changes can be automated (as with our automated `ptr<T>` conversion tool) or made unnecessary in the future by relaxing requirements such as the additions of initializers.

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 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.

Layout Changes 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.

6.3 Performance Results

An important concern about run-time checking for C is the effect on performance and compile time. 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 (meaning performance is within 2% of) 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.

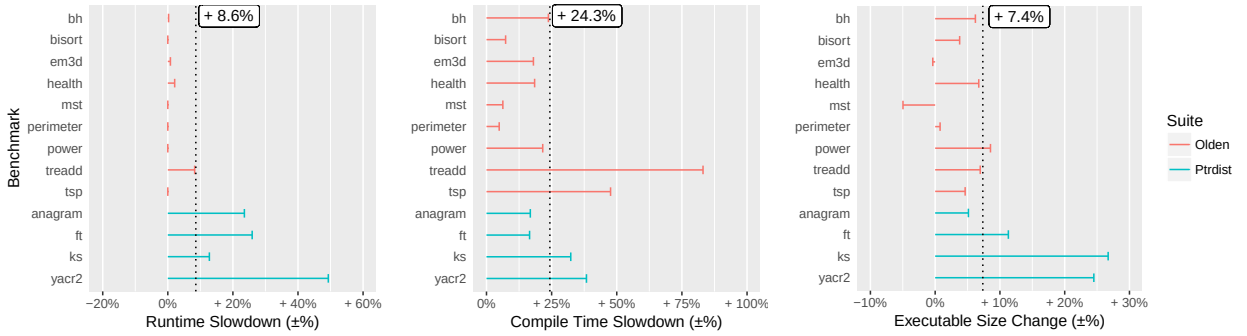

Figure 6.2: Performance Overheads

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 have spent no time at all optimising compile-time for Checked C. In particular, treeadd is a major outlier because the program is so short.

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%.

6.4 Evaluation

All considered, we offer equivalent or better performance than either CCured or Deputy provide, at the outlay of having to convert a larger amount of code. We believe this to be a good trade-off, partially because we feel most of these changes can be automated, and partially because we provide the programmer greater control over how they express their bounds in terms of other program data. The latter in particular allows the programmer to work with their optimizer to ensure run-time overhead, especially in tight loops, remains low.

6.4.1 Interaction with the LLVM Optimizer

Currently we have done no specific work within our compiler to elide or optimize our dynamic checks. Any checks that Clang/LLVM currently removes, it does so using its existing optimization passes.

We expected the LLVM optimizer to be a lot worse at optimizing dynamic checks than it turned out to be. In only two of our benchmarks did we have to hand-optimize these programs to reduce the overhead of dynamic checking to approximately 0%. This was accomplished by a mixture of being more specific with annotated bounds (including introducing temporary variables) and hoisting checks from inner loops using explicit dynamic checks.

Expressions that caused most difficulty for removing dynamic checks was code involving (at least) two pointer dereferences, as this incurs an unavoidable non-null check on the inner pointer, even if the outer pointer can be proven to be non-null. This can cause overhead in loops iterating through, for example, linked lists or graphs. We are working on a proposal to add nullness (and non-null) annotations to pointers in Checked C, which would allow for these inner non-null checks to be elided in many cases.

Chapter 7

Related Work

Attempting to make C memory safe is a well-trodden path [19]. Approaches fall into four categories: better type systems; runtime representations of bounds; static analysis annotations; and instrumentation. We avoid mentioning approaches that propose new languages or operating-system integration of protections.

Better Type Systems In this approach, the compiler adds new pointer types, perhaps including bounds information, with rules that are enforced at compile time either by the type-checker or by other static analyses. Any found errors prevent compilation from being successful.

The largest influence on Checked C's approach is Deputy [2], a dependent-type system for C. This project used dependent types for two reasons, the first is to represent array bounds, the second is to ensure access to unions is via the correct member. Unfortunately, to use Deputy, the programmer had to rewrite their C program all at once, which is uneconomical and raises the effort required to get any safety payoff. With our unchecked blocks and interoperation types, we do not require the whole program to be translated at once. The programmer also has little to no control over the bounds expressions, preventing dynamic check optimizations.

CCured [12] takes a similar approach to ours, with new pointer types to represent various kinds of arrays. They couple this with an analysis of pointer usage which we have emulated in the conversion tool described in [16].

Run-time Representations of Bounds In this approach, the compiler uses extra memory to represent run-time bounds. This then allows the compiler to insert checks which ensure pointers are within bounds. They may also have to generate code to update bounds information as other data within the program changes.

For instance, the SoftBound [11] work has a shadow memory space where they store information about pointer bounds. In order to achieve low run-time overhead, this approach requires up to twice the memory of the original program, something that is not always available on all platforms. Other work in the same vein [10] uses hardware-based approaches that require Memory Management Unit support.

CCured [12] also uses run-time representations for its pointers, changing the in-memory layout, which we wanted to avoid. In particular, pointer values are now represented by up to three pointers, one for the pointer itself, and one each for its upper and lower bounds. CCured also uses a garbage collector to implement sound management of pointers it cannot infer information about, something we have avoided the runtime overhead of.

Static Analysis Annotations In this approach, the programmer adds manual annotations to pointer declarations, similar to our bounds expressions. These are then checked by an external static analyser. The main difference is correct usage of these is enforced by separate static analysis, and is not built into the original compiler.

The main solution in this space is SAL [9], which can detect, but not prevent bugs. Its annotations not only denote bounds, but also whether a pointer may be read from or written to. SAL includes annotations for function behaviour, such as locking, which we do not include.

Instrumentation Lastly, we have some dynamic analysis systems that are not intended for production usage, but can detect incorrect usage of the C language. They are intended for usage in testing setups in conjunction with fuzzing systems that ensure the dynamic analysis can reach most program points.

The two most prominent of these are ASan [17] and UBSan [8]. These can detect and prevent addressing and undefined behaviour issues in C dynamically, including out-of-bounds accesses and invalid pointer arithmetic.

Chapter 8

Conclusion

This report has described how I implemented the dynamic checks in the Checked C compiler. I have presented a flow-insensitive algorithm for propagating bounds information through expressions, such that these bounds can be checked when memory is accessed, and has shown one possible compilation of these bounds expressions into dynamic checks.

I have also shown that, for a set of pointer-intensive benchmarks, though the programmer will have to modify on average 17.5% of each benchmark in order to have the soundness Checked C guarantees, this gives a run-time overhead of on average 8.6% and at most 49.3%.

This work is an important step towards proving the viability and usefulness of Checked C.

Appendix A

Bibliography

[1] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. *SIGPLAN Not.*, 29(6):290–301, June 1994.

[2] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proceedings of European Symposium on Programming (ESOP '07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 520–535, Heidelberg, 2007. Springer-Verlag.

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

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

[5] Stephen Kell. Some were meant for C: The endurance of an unmanageable language. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 229–245, New York, NY, USA, 2017. ACM.

[6] Robbert Krebbers and Freek Wiedijk. Subtleties of the ANSI/ISO C standard. Document N1637, ISO/IEC JTC1/SC22/WG14, September 2012.

[7] LLVM. *LLVM IR Language Reference*. `http://llvm.org/docs/LangRef.html`, Accessed 29 October 2017.

[8] LLVM. *Undefined Behavior Sanitizer*. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`, Accessed 5 November 2017.

[9] Microsoft. *Using SAL Annotations to Reduce C/C++ Code Defects*. `https://docs.microsoft.com/en-us/visualstudio/code-quality/using-sal-annotations-to-reduce-c-cpp-code-defects`, Accessed 5 November 2017.

[10] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Everything You Want to Know About Pointer-Based Checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 190–208, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[11] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve 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*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

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

[13] NIST vulnerability database. `https://nvd.nist.gov`. Accessed May 17, 2017.

[14] Dennis C. Ritchie and Brian W. Kernighan. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 2nd edition, 1988.

[15] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. Program. Lang. Syst.*, 17(2):233–263, March 1995.

[16] Andrew Ruef, Archibald Samuel Elliott, David Tarditi, and Michael Hicks. Checked C for safety, gradually. Draft Paper `http://lenary.co.uk/publications/checkedc_gradually/`, May 2017.

[17] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[18] Synopsys, Inc. The heartbleed bug. `http://heartbleed.com`, April 2014. Accessed Oct 17, 2017.

[19] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[20] David Tarditi. *Extending C with bounds safety.* Microsoft, January 2017. Version 0.6 `https://github.com/Microsoft/checkedc/releases/tag/v0.6-final`.

[21] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *ESOP*, 2009.

[22] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *7th Symposium on Operating System Design and Implementation (OSDI'06)*, Seattle, Washington, 2006. USENIX Association.

Appendix B

End Notes

1. In fact, in C, no function parameter has array type, as you cannot pass an array (checked or not) to a function. Arrays are passed by passing a pointer to their first element, so C functions that are declared to have array-typed parameters actually have the equivalent pointer-typed parameter. Checked C does the same with checked arrays and checked pointers to arrays respectively.

2. This requires that the Checked C code in that checked region does not perform any unsound casts, which is a hard property to formalize. Importantly we need to prevent casts between objects of two incompatible types, even if this is done by casting via a type that is compatible with both of these types, such as `void*`.

3. What the C11 Specification calls "values" are more commonly called "r-values" in other language descriptions.

4. In C, the `NULL` pointer does not point to any object, and you may not use a pointer into one object to compute the pointer into another object unless the former contains the latter object. Therefore computing a pointer offset from `NULL` is very much undefined behaviour. LLVM matches this by allowing computed offsets from `null` to be optimized to `null`.

Appendix C

C Syntax

This appendix contains a definition of the relevant parts of C Syntax for this report. This is based on the C11 Specification, but written with mathematical symbols rather than ASCII text.

LValue Expressions

$e_l ::= x$	Variables
$\mid *e_v$	Dereference
$\mid e_v[e_v]$	Index
$\mid e_l.m$	Struct Member Access
$\mid e_v \rightarrow m$	Pointer Member Access

Value Expressions

$e_v ::= \&e_l$	Address-of
$\mid e_l = e_v$	Assignment
$\mid e_l \oplus= e_v$	Compound Assignment
$\mid e_l{+}{+} \mid {+}{+}e_l$	Increment
$\mid e_l{-}{-} \mid {-}{-}e_l$	Decrement
$\mid e_v.m$	Struct Member Access
$\mid e_v, e_v$	Comma Expression
$\mid e_v \oplus e_v$	Binary Operation
$\mid e_v \ R \ e_v$	Binary Relational Operation
$\mid \otimes e_v$	Unary Operation
$\mid e_v(\overline{e_v})$	Function Call
$\mid (t)e_v$	Cast
$\mid e_v \ ? \ e_v : e_v$	Conditional Operator
$\mid l$	Literal Value
$\mid e_l$	Lvalue & Array Conversion

Binary Operators

$\oplus ::= +$	Addition
$\mid -$	Subtraction
$\mid *$	Multiplication
$\mid /$	Division
$\mid \%$	Modulus
$\mid \&$	Bitwise AND
$\mid \mid$	Bitwise OR
$\mid \wedge$	Bitwise XOR
$\mid \gg$	Bitwise Shift Right
$\mid \ll$	Bitwise Shift Left

Relational Operators

$R ::= \&\&$	Logical AND
$\mid \mid\mid$	Logical OR
$\mid ==$	Equality
$\mid !=$	Negated Equality
$\mid > \mid \geq \mid < \mid \leq$	Comparison

Unary Operators

$\otimes ::= +$	Unary Plus
$\mid -$	Numerical Negation
$\mid \sim$	Bitwise Negation
$\mid !$	Logical Negation

Literals

$l ::=$ `NULL`	Null Pointer Literal
$\mid$ integers	Integer Literals
$\mid$ floats	Float Literals
$\mid \{l, \dots\}$	Array & Struct Literals

Types

$t ::= \dots$

Struct Members

$m ::= \dots$

Appendix D

Propagation Rules

These are the algebraic definitions of our propagation algorithm to go with the descriptions in Section 4.3. Certain helper functions are explained in the text, rather than algebraically.

$$
\begin{aligned}
\textit{lvalue-bounds}(e_l) &\in \text{bounds}(e_v, e_v) && \text{Lvalue Bounds} \\
\textit{lvalue-bounds}(x) &= \textit{array-bounds}(x, \texttt{N}) && \text{Variables} \\
&\quad \text{when } \text{type}(x) = \text{array}(\texttt{T}, \texttt{N}) \\
&= \textit{single-bounds}(\&x) \text{ otherwise.} \\
\textit{lvalue-bounds}(*e_v) &= \textit{value-bounds}(e_v) && \text{Dereference} \\
\textit{lvalue-bounds}(e_1[e_2]) &= \textit{value-bounds}(e_{base}) && \text{Index} \\
&\quad \text{where } e_{base} = \textit{base}(e_1, e_2) \\
\textit{lvalue-bounds}(e_l.m) &= \textit{array-bounds}(e_l.m, \texttt{N}) && \text{Struct Member Access} \\
&\quad \text{when } \text{type}(e_l.m) = \text{array}(\texttt{T}, \texttt{N}) \\
&= \textit{single-bounds}(\&(e_l.m)) \text{ otherwise.} \\
\textit{lvalue-bounds}(e_v{\to}m) &= \textit{array-bounds}(e_v{\to}m, \texttt{N}) && \text{Pointer Member Access} \\
&\quad \text{when } \text{type}(e_v{\to}m) = \text{array}(\texttt{T}, \texttt{N}) \\
&= \textit{single-bounds}(\&(e_v{\to}m)) \text{ otherwise.} \\
\textit{single-bounds}(e_v) &= \text{bounds}(e_v, e_v + 1) && \text{Single Object Bounds} \\
\textit{array-bounds}(e_l, N) &= \text{bounds}(e_l, e_l + N) && \text{Constant-sized Array Bounds} \\
&\quad \text{when } N \text{ is constant.} \\
&= \text{bounds}(\text{unknown}) \text{ otherwise.}
\end{aligned}
$$

Figure D.1: Lvalue Bounds Propagation Rules (Mutually Recursive with Figure D.2)

$$\textit{value-bounds}(e_v) \in \text{bounds}(e_v, e_v) \qquad \text{Value Bounds}$$

$$\textit{value-bounds}(\texttt{NULL}) = \text{bounds}(\text{any}) \qquad \text{Null Pointer Literal}$$

$$\textit{value-bounds}(l) = \text{bounds}(\text{unknown}) \qquad \text{Other Literals}$$

$$\textit{value-bounds}(\&e_l) = \textit{lvalue-bounds}(e_l) \qquad \text{Address-of}$$

$$\textit{value-bounds}(e_l = e_v) = \textit{value-bounds}(e_v) \qquad \text{Assignment}$$

$$\textit{value-bounds}(e_l \oplus= e_v) = \textit{lvalue-target-bounds}(e_l) \qquad \text{Compound Assignment}$$

$$\textit{value-bounds}(e_l{+}{+}) = \textit{lvalue-target-bounds}(e_l) \qquad \text{Increment \& Decrement}$$

Likewise for all Increment & Decrement operators

$$\textit{value-bounds}(e_v.m) = \textit{struct-rel-bounds}(e_v, m) \qquad \text{Struct Member Operator}$$

$$\textit{value-bounds}(e_f(\overline{e_{arg}})) = \textit{declared-return-bounds}(e_f)\,[\text{params}(f)/\overline{e_{arg}}] \qquad \text{Function Call}$$

$$\textit{value-bounds}(e_{v1}, e_{v2}) = \textit{value-bounds}(e_{v2}) \qquad \text{Comma Expression}$$

$$\textit{value-bounds}(e_l) = \textit{lvalue-bounds}(e_l) \qquad \text{Array Conversion}$$

when $\text{type}(e_l) = \text{array}(\mathit{T}, \mathit{N})$

$$= \textit{lvalue-target-bounds}(e_l) \text{ otherwise} \qquad \text{Lvalue Conversion}$$

$$\textit{value-bounds}(e_1 \oplus e_2) = \textit{value-bounds}(e_{base}) \qquad \text{Pointer Arithmetic}$$

when $e_1 \oplus e_2$ has pointer type

where $e_{base} = \mathit{base}(e_1, e_2)$

$$\textit{value-bounds}(e_1\ R\ e_2) = \text{bounds}(\text{unknown}) \qquad \text{Binary Relational Operators}$$

$$\textit{value-bounds}(!e_v) = \text{bounds}(\text{unknown}) \qquad \text{Logical Negation Operator}$$

$$\textit{value-bounds}(e_v) = \textit{value-bounds}(e_{vi}) \qquad \text{Other Expressions}$$

where e_{vi} is the subexpression of e_v with known bounds

If more than one subexpression of e_v has known bounds,

all calculated bounds must be equal.

$$= \text{bounds}(\text{unknown}) \text{ otherwise}$$

$$\textit{declared-return-bounds}(f) \in \text{bounds}(e_v, e_v) \qquad \text{Bounds on Function Return}$$

$$\textit{struct-rel-bounds}(e_v, m) \in \text{bounds}(e'_v, e'_v) \qquad \text{Struct Relative Bounds}$$

Figure D.2: Value Bounds Propagation Rules (Mutually Recursive with Figure D.1 and Figure D.3)

$$\textit{lvalue-target-bounds}(e_l) \in \text{bounds}(e_v, e_v) \qquad \text{Lvalue Target Bounds}$$

$$\textit{lvalue-target-bounds}(e_l) = \textit{single-bounds}(e_l) \quad \text{where } \text{type}(e_l) = \texttt{ptr<}T\texttt{>} \qquad \text{Checked Singleton Pointer Bounds}$$

$$\textit{lvalue-target-bounds}(x) = \textit{declared-bounds}(x) \qquad \text{Variable Target Bounds}$$

$$\textit{lvalue-target-bounds}(e_l.m) = \textit{struct-rel-bounds}(e_l, m) \qquad \text{Struct Member Target Bounds}$$

$$\textit{lvalue-target-bounds}(e_v \rightarrow m) = \textit{struct-rel-bounds}(e_v, m) \qquad \text{Pointer Member Target Bounds}$$

$$\textit{lvalue-target-bounds}(\_) = \text{bounds}(\text{unknown})$$

$$\textit{declared-bounds}(x) \in \text{bounds}(e_v, e_v) \qquad \text{Declared Bounds}$$

Figure D.3: Lvalue Target Bounds Propagation Rules