Checked C: Adding Memory Safety to LLVM

Mandeep Singh Grang
Katherine Kjeer
WHAT WE'LL DISCUSS

• What is Checked C?
• Implementation of Checked C in Clang
• Novel algorithm to widen bounds for null-terminated pointers
• Novel algorithm for comparison of expressions
• Conversion of legacy C code to Checked C
• Experimental evaluation
• Resources
MEMORY SAFETY HAZARDS IN C

Buffer Overflow

Null Pointer Dereference

4 Bytes Buffer | 4 Bytes Overflow

Points to Nothing

O_0^ V_1^ E_2^ R_3^ F_4^ L_5^ O_6^ W_7^
CHECKED C: IN A NUTSHELL

**Extension to C**
Supports spatial safety

**New Pointer Types**
Adds 3 new pointer types that are bounds-checked

**Incremental Porting**
Allows incremental porting from legacy C

**Syntax like C++**
Syntax for checked pointers is borrowed from C++ templates

**Implemented in Clang**
Checked C has been implemented in our fork of Clang
Points to a Single Object
Points to an object of type T

No Pointer Arithmetic
Pointer used for dereference only

Runtime Check for Non-nullness
Non-nullness checked at runtime, if necessary
<table>
<thead>
<tr>
<th>C</th>
<th>Checked C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T *x;</code></td>
<td><code>_Ptr&lt;T&gt; x;</code></td>
</tr>
<tr>
<td><code>int *p;</code></td>
<td><code>_Ptr&lt;int&gt; p;</code></td>
</tr>
<tr>
<td><code>const int *p;</code></td>
<td><code>_Ptr&lt;const int&gt; p;</code></td>
</tr>
<tr>
<td><code>int x;</code></td>
<td><code>int x;</code></td>
</tr>
<tr>
<td><code>int *const p = &amp;x;</code></td>
<td><code>const _Ptr&lt;int&gt; p = &amp;x;</code></td>
</tr>
</tbody>
</table>
[] Pointer to Array
Points to an element of an array of type T

[[ ]] Pointer Arithmetic Allowed
Pointer arithmetic can be done on this pointer type

Runtime Check for Bounds
Non-nullness and bounds checked at runtime, if necessary
<table>
<thead>
<tr>
<th>C</th>
<th>Checked C</th>
</tr>
</thead>
<tbody>
<tr>
<td>T *x = &quot;&quot;; T x[] = {};</td>
<td>_Array_ptr&lt;T&gt; x = &quot;&quot;; T x _Checked[] = {};</td>
</tr>
<tr>
<td>const char *p = &quot;abc&quot;;</td>
<td>_Array_ptr&lt;const char&gt; p = &quot;abc&quot;;</td>
</tr>
<tr>
<td>char *foo(char p[]);</td>
<td>_Array_ptr&lt;char&gt; foo(char p _Checked[]);</td>
</tr>
</tbody>
</table>
Null Terminated Array
Points to a sequence of elements that ends with a null terminator

Element Access
An element of the sequence can be read provided the preceding elements are not the null terminator

Automatic Bounds Widening
Bounds can be widened based on number of elements read
<table>
<thead>
<tr>
<th>C</th>
<th>Checked C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T *x = &quot;&quot;;</code></td>
<td><code>_Nt_array_ptr&lt;T&gt; x = &quot;&quot;;</code></td>
</tr>
<tr>
<td><code>T x[] = {};</code></td>
<td><code>T x _Nt_checked[] = {};</code></td>
</tr>
<tr>
<td><code>const char *p = &quot;abc&quot;;</code></td>
<td><code>_Nt_array_ptr&lt;const char&gt; p = &quot;abc&quot;;</code></td>
</tr>
<tr>
<td><code>char *foo(char p[]);</code></td>
<td><code>_Nt_array_ptr&lt;char&gt; foo(char p _Nt_checked[]);</code></td>
</tr>
</tbody>
</table>
LIMIT MEMORY
Describe memory range pointer can access

LOW-LEVEL CONTROL
Programmer declares bounds that act as invariants

RUNTIME CHECKS
Check that memory accesses are within bounds

STATIC CHECKS
Check that bounds invariants are not violated

**COUNT**

\[ p : \text{count}(n) \]

\( p \) can access \( n \) array elements.

**BYTE COUNT**

\[ p : \text{byte\_count}(n) \]

\( p \) can access \( n \) bytes.

**RANGE**

\[ p : \text{bounds}(e_1, e_2) \]

\( p \) can access memory from \( e_1 \) to \( e_2 \).

**UNKNOWN**

\[ p : \text{bounds}(\text{unknown}) \]

\( p \) cannot be used to access memory.
void f(_Array_ptr<int> p : count(len), size_t len) {
    for (int i = 0; i <= len; ++i) {
        int n = p[i];
    }
}
void f(_Array_ptr<int> p : count(len),
    size_t len) {
    for (int i = 0; i <= len; ++i) {
        int n = p[i];
    }
}
void f(_Array_ptr<int> p : count(len),
  size_t len) {
  for (int i = 0; i <= len; ++i) {
    int n = p[i];
  }
}
void f(_Array_ptr<int> p : count(len), size_t len) {
    for (int i = 0; i <= len; ++i) {
        int n = p[i];
    }
}
void f(_Array_ptr<int> p : count(len), size_t len) {
    for (int i = 0; i <= len; ++i) {
        int n = p[i];
    }
}
void f(_Array_ptr<int> p : count(len),
    size_t len) {
  for (int i = 0; i <= len; ++i) {
    int n = p[i];
  }
}

When i == len:
runtime error!

1. Visit an expression that reads memory via a pointer
   p[i]

2. Get the pointer-typed expression that reads memory
   p[i] == *(p + i) => p + i (pointer p)

3. Get the bounds of the pointer-typed expression
   count(len) => bounds(p, p + len)

4. Insert a dynamic check for the expression and bounds
   p + i, bounds(p, p + len)

5. At runtime, check that the pointer is within bounds
   0 <= (p + i) < (p + len)
   0 <= i < len
STATICALLY CHECKING BOUNDS DECLARATIONS

**INFER**
Bounds for pointer-typed expressions

**CONVERT**
Inferred and declared bounds to ranges

**CHECK**
Declared range is within inferred range

**ASSIGNMENTS:**
Check RHS bounds contain LHS bounds

**FUNCTION CALLS:**
Check arg bounds contain param bounds

LHS = RHS;
void f(param);
f(arg);
void f(_Array_ptr<int> p : count(x), int x, _Array_ptr<int> q : count(3)) {
    p = q;
    p = (int _Checked[]){ 0, 1 };
    p++; // Original value of p: p - 1.
    x = x * 2; // No original value for x.
}

<table>
<thead>
<tr>
<th>Declared</th>
<th>Inferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p, p + x)</td>
<td>(q, q + 3)</td>
</tr>
<tr>
<td>p = q;</td>
<td>{ 0, 1 }, { 0, 1 } + 2</td>
</tr>
<tr>
<td>p = (int _Checked[]){ 0, 1 }</td>
<td></td>
</tr>
<tr>
<td>p++; // Original value of p: p - 1.</td>
<td>(p - 1, p - 1 + x)</td>
</tr>
<tr>
<td>x = x * 2; // No original value for x.</td>
<td>unknown</td>
</tr>
</tbody>
</table>
CONVERT BOUNDS TO RANGE

Lower Bound: (p - 2, p + 3)

Base: p

Lower Offset: -2

Upper Offset: 3
CHECK RANGES
_Array_ptr<char> p : count(2) = 0;
(Array_ptr<char> q : count(1) = 0;
// Error: declared bounds for 'p' are invalid after assignment.
p = q;

Declared

Inferred
_Array_ptr<char> p : count(2) = 0;
_Array_ptr<char> q : count(e) = 0;
// Warning: cannot prove declared bounds for 'p'
// are valid after assignment.
p = q;
_Nt_array_ptr<T> p : bounds(p, p) = "";

if (*p)
  // Ptr deref is at upper bound. Widen the bounds. New bounds: (p, p + 1)

if (*(p + 1))
  // Ptr deref is at upper bound. Widen the bounds. New bounds: (p, p + 2)

if (*(p + 3))
  // Ptr deref is NOT at upper bound. No bounds widening. Flag ERROR!
  error: out-of-bounds memory access: if (*(p + 3))
  note: accesses memory at or above the upper bound
  note: inferred bounds are 'bounds(p, p + 2)'
Forward
A basic block is visited before its successors

Path-Sensitive
Dataflow analysis generates different facts on the *then* and *else* branches

Flow-Sensitive
Dataflow analysis depends on the order of statements in a basic block

Intra-Procedural
Dataflow analysis is done on one function at a time

**BOUNDS WIDENING**

**DATAFLOW EQUATIONS**

**Init:**
- **In[Bi->Bj]**
  - \( \cap \text{Out[Bi->Bj]} \), where \( Bi \in \text{pred}(B) \)
  - **Init:** \( \text{In[Entry]} = \emptyset \)
  - **In[B] = Top**

- **Out[Bi->Bj]**
  - \( (\text{In[Bi]} - \text{Kill[Bi]}) \cup \text{Gen[Bi->Bj]} \)
  - **Init:** \( \text{Out[Entry->Bj]} = \emptyset \)
  - **Out[Bi->Bj] = Top**

**Gen[Bi->Bj]**
- \( \text{Gen[Bi->Bj]} \cup \{p:1\} \)
  - where \( p \in _Nt_array_ptr \) and \( i \in \text{decl_bounds}(p) \)

**Kill[B]**
- \( \text{Kill[B]} \cup \{p\} \)
  - where \( p \in _Nt_array_ptr \) and \( i \in \text{decl_bounds}(p) \)
1: int k = 0;
2: _Nt_array_ptr<T> p : bounds(p, p + k);
3: while (*p + k))
4: if (*p + k + 1))
5: k = 42;
Should Widen Bounds

- if (*(p + i + j + 1 + 3))
- if (*(2 + i + p + j + 2 + 0))
- if (*(p + 5 + i - 1 + j))
- if (*(j + p + i + (2 * 2)))

Should Not Widen Bounds

- if (*(p + i + j + 3))
- if (*(p + (i * j) + 4))
- if *(p + i + 4))
- if *(p + i + j + 4 + k))

"We need a mechanism to determine if two expressions are equivalent"
**N-ary**

The preorder AST is an n-ary tree.

**Preorder**

It represents an expression in the preorder form.

**Flattened**

The tree is flattened at each level by coalescing nodes with their parents.

**Normalized**

The underlying expression is normalized by constant-folding and sorting the nodes of the tree.

SEMANTIC COMPARISON OF EXPRESSIONS USING THE PREORDER AST

Are E1 and E2 equivalent?

Step 1: Create preorder ASTs for E1 and E2

E1 = (b + c + a) * (e + 3 + d + 5)
E2 = (2 + 3 + 3 + d + e) * (c + a + b)
Step 2: Coalesce leaf nodes having a commutative and associative operator
SEMANTIC COMPARISON OF EXPRESSIONS USING THE PREORDER AST

Step 2: Coalesce leaf nodes having a commutative and associative operator
Step 2: Coalesce leaf nodes having a commutative and associative operator
Step 2: Coalesce leaf nodes having a commutative and associative operator
Step 3: Lexicographically sort leaf nodes having a commutative and associative operator.
Step 3: Lexicographically sort leaf nodes having a commutative and associative operator.
Step 4: Constant-fold integer nodes having a commutative and associative operator.
Step 4: Constant-fold integer nodes having a commutative and associative operator
Step 5: Sort subtrees having a commutative and associative operator.
Step 5: Sort subtrees having a commutative and associative operator.

Now compare the two ASTs node-by-node to check if the underlying expressions are equivalent.
Integer overflow due to re-association of expressions

\[(e1 + e2) + e3\]  
Original expression may not overflow

\[e1 + (e2 + e3)\]  
Expression may overflow after re-association!

Possible Solution

`-fwrapv`  
Treat signed integer overflow as two's complement

What about pointer arithmetic overflow?

`-fwrapv-pointer`  
GCC has this flag
INCREMENTAL CONVERSION

Unchecked

Checked
Conversion without breaking compatibility?

[ ] Bounds-safe interface
BOUNDS-SAFE INTERFACES

CONVERSION SUPPORT
Port from legacy C a few lines at a time

ALTERNATE TYPES
Specify types for checked parameters

OPTIONAL BOUNDS
Checked arguments must meet bounds

BACKWARDS COMPATIBLE
Accept unchecked pointer arguments

BOUNDS CHECKING
Check bounds for checked arguments

https://bit.ly/35qXht0
CONVERTING A FUNCTION

1. Determine checked types
2. Add parameter and return bounds
3. Update function calls
char *strncpy(char *dest, char *src, size_t n);
char *strncpy(
    char *dest : itype(_Nt_array_ptr<char>),
    char *src : itype(_Nt_array_ptr<char>),
    size_t n
): itype(_Nt_array_ptr<char>);
// strncpy copies the first n characters of src into dest.
char *strncpy(
    char *dest : itype(_Nt_array_ptr<char>) count(n),
    char *src : itype(_Nt_array_ptr<char>) count(n),
    size_t n
) : itype(_Nt_array_ptr<char>) count(n);
void unchecked_pointers() {
    // dest points to 3 characters including null terminator.
    char *dest = "12\0";
    // src points to 2 characters including null terminator.
    char *src = "1\0";
    // Fine - there is no bounds checking for dest and src.
    strncpy(dest, src, 3);
}

void checked_pointers() {
    // dest points to 3 characters including null terminator.
    _Nt_array_ptr<char> dest : count(3) = "12\0";
    // src points to 2 characters including null terminator.
    _Nt_array_ptr<char> src : count(2) = "1\0";
    // Fine – dest and src both point to at least 2 characters.
    strncpy(dest, src, 2);
    // Error: src points to 2 characters, expected to
    // point to at least 3.
    strncpy(dest, src, 3);
}
CHALLENGE: STRING LENGTHS

char *strupr(char *str);

char *strupr(char *str : itype(_Nt_array_ptr<char>));

char *strupr(char *str : itype(_Nt_array_ptr<char>) count(?));
This would be great...

```c
char *strupr(
    char *str : itype(_Nt_array_ptr<char>)
    count(strlen(str))
);
```

...but it's not possible
WHY CAN'T WE USE STRLEN?

No modifying expressions are allowed in bounds

Function calls may modify memory

```c
char *strupper(
    char *str : itype(_Nt_array_ptr<char>)
    count(len),
    size_t len
);
```
CHECKEDC-CONVERT

CONVERT POINTERS
_Ptr<T>
_T * _Array_ptr<T>
_Nt_array_ptr<T>

UMD
Developed at the University of Maryland
CONVERTING MUSL

2 interns
5 weeks
"abc"

string subdirectory
network subdirectory
MUSL STRING LIBRARY

31 functions converted
72 total functions

316 lines of code converted
1574 total lines of code
51 FUNCTIONS CONVERTED

729 LINES OF CODE CONVERTED

65 TOTAL FUNCTIONS

3524 TOTAL LINES OF CODE
EVALUATION

LNT TESTS
Olden and Ptrdist benchmarks

CODE SIZE
Impact on generated code

RUNTIME
Overhead introduced by dynamic checks

COMPILING
Impact on compilation time
17.5% Average LOC modified
9.3% Average unchecked

REMAINING UNCHECKED CODE

% Code still unchecked after conversion
7.4% Average overhead
Lower is better
8.6% Average overhead

Lower is better
24.3% Average overhead

Lower is better
Code Repository

Language Specification
https://bit.ly/2FmPyRO

SecDev 2018 Paper
https://bit.ly/2Zt2k8g