PLDI
San Diego 2022

Tutorial: Making C Programs Safer with Checked C

Jie Zhou
University of Rochester

Yudi Yang
University of Rochester

Michael Hicks
*University of Maryland & Amazon**

John Criswell
University of Rochester

**work on Checked C done prior to starting at Amazon*

Before We Start

Download the Git repo and Docker image:

- https://github.com/URSec/CheckedC-Tutorial-PLDI22
- (Follow the instructions to install Docker)
- Download and start the docker container:
  - `./run.sh` on Linux/MacOS
  - Double-click `run.bat` on Windows

Compiler used in this tutorial: commit `107dee3`

`https://github.com/secure-sw-dev/checkedc-llvm-project`

What We (Systems Programmers) Want

Make C Programs Safe

- Static/dynamic analysis—false positives/negatives
- Compiler-based transformation—high overhead & poor programmer control
- New PL, e.g., Rust—good for new programs but not for curing legacy code

A More Realistic Goal

Make C Programs *Safer* with Checked C

Today's Goal:

- Introducing ***Checked C***: a new safe extension to C
- Introducing ***3C***: a semi-automatic tool that converts C to Checked C

Outline

- Part I: Checked C
  - Technical Overview (30–35 minutes)
  - (10 minute break)
  - Hands-on Practice: Write Checked C code (30–40 minutes)

30 minute break

- Part 2: Semi-automated porting with 3C
  - Technical Overview and Demo of 3C (40 minutes)
  - (10 minute break)
  - Hands-on Practice: Use 3C to port legacy C code (25 minutes)

Checked C

- Originating from Microsoft Research, now maintained by the Secure Software Development Project (https://github.com/secure-sw-dev)
- A new *extension* to C, aiming for **memory safety** and **type safety**
- Open-sourced from its inception

Design Principles of Checked C

High performance

- New checked pointer types and strong static analysis

Good human readability and long-term maintainability

- Explicit checked pointers and bounds information of memory objects

Easy incremental porting legacy C code

- Mixing checked and legacy C code with fine granularity
- Good backward compatibility *via bounds-safe interfaces*

A new *extension* to C, aiming for **memory safety** and **type safety**

Scope of Checked C

Currently focuses on ***spatial*** memory safety (topic of today)

- Out-of-bounds memory access
- Null pointer dereference

Ongoing research work for ***temporal*** memory safety (not covered today)

- Use-After-Free (UAF)
- Double free
- Invalid free

Checked Pointers

Three new pointer types:

- `_Ptr`<T>: pointer to a singleton object of type T
- `_Array_ptr`<T>: pointer to an array of type T
- `_Nt_array_ptr`<T>: pointer to a null-terminated (ends with `'\0'`) array of type T

Checked Pointer: _Ptr<T>

Pointing to a single memory object of type T — *no pointer arithmetic or subscript operator (e.g., x[0])*

```
struct Data {
    int val;
    long lval;
    ...
};

_Ptr<struct Data> p = malloc(sizeof(struct Data));
...
...
printf("val = %d\n", p->val);
```

→ *null-pointer check inserted by compiler, if not determined at compile time*

```
...
...
p++;
```

→

```
error: arithmetic on _Ptr type
        p++;
        ~~^
```

Checked Pointer: `_Array_ptr<T>`

Pointing to an array of object of type T — *permits pointer arithmetic and subscripting*

- bounds declared by programmers
- bounds checking inserted by compiler when not provable at compile time

```
_Array_ptr<int> p = malloc(sizeof(int) * BUF_LEN);

int i = p[5];
```



```
error: expression has unknown bounds
    int i = p[5];
            ^~~
```

Bounds Declaration for _Array_ptr<T>

```
_Array_ptr<T> p : count(bounds_expr) = …;
```

```
#define BUF_LEN 30
_Array_ptr<int> p : count(BUF_LEN) = malloc(sizeof(int) * BUF_LEN);
int i = p[5];
```

no dynamic check inserted because compiler knows BUF_LEN > 5

```
int j = p[30]
```

error: out-of-bounds memory access

```
                int j = p[30];
                        ^~~~
int k = p[var];
```

null-pointer and bounds checking inserted by compiler, if `var < BUF_LEN` *is not provable*

```
_Array_ptr<int> p : count(BUF_LEN / 2) = malloc(sizeof(int) * BUF_LEN);
```

allowed

```
int j = p[15]
```

error: out-of-bounds memory access

```
                int j = p[15];
                        ^~~~
```

because p's bounds is `[p, p + 14)`

```
_Array_ptr<int> p : count(BUF_LEN + 1) = malloc(sizeof(int) * BUF_LEN);
```

error: declared bounds for 'p' are invalid after initialization

Bounds Declaration for _Array_ptr<T>

```
_Array_ptr<T> p : bounds(lower_b, upper_b) = …;
```

```
_Array_ptr<int> p0 : count(BUF_LEN) = malloc(sizeof(int) * BUF_LEN);

_Array_ptr<int> p1 : bounds(p0, p0 + BUF_LEN / 2) = p0;

int i = p1[15]
```

```
error: out-of-bounds memory access
        int i = p1[15];
                 ^~~~~
```

Checked Pointer: _Nt_array_ptr<T>

Pointing to a null-terminated array of type T (integral and pointer type) — *allowing pointer arithmetic*

```
_Nt_array_ptr<char> p0 : count(5) = “12345";
```

p0 actually has bounds(p0, p0 + 6). p0[5] (‘\0’) is legal.

```
_Nt_array_ptr<char> p = "12345";
```

equivalent to: “p : bounds[p, p + 1)”

```
char c = p[0]; legal
char c = p[1];
```

```
error: out-of-bounds memory access
    char c = p[1];
             ^~~~
```

Automatic bounds widening

```
if (*p == '1') { legal
    if (*(p + 1) == '2') {  legal, because p's bounds has been widened to [p, p + 2)
        if (*(p + 3) == '3') {
            ...
        }
    }
}
```

```
error: out-of-bounds memory access
```

because p’s bounds is [p, p + 3)

Checked Regions

Checked C allows mixing checked and original C code with *fine granularity*.

New keyword: `_Checked`

- *Annotating* a block of code, from a single statement to a whole source file
- Enforcing *more strict typing rules*, e.g., only checked pointers allowed
- Helping programmers to *narrow down* the scope of memory safety bugs:
  Checked regions are provably blameless of causing spatial memory safety errors[1]

`_Unchecked` regions: Force the compiler to omit checking

```
#pragma CHECKED_SCOPE on
```

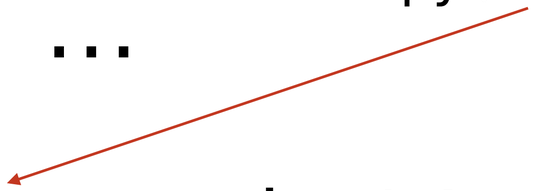
All code below it is in checked region, unless turned off by `#pragma CHECKED_SCOPE off`

[1] **A Formal Model of Checked C**. Liyi Li, Yiyun Liu, Deena L. Postol, Leonidas Lampropoulos, David Van Horn, and Michael Hicks. CSF'22

Bounds-safe Interfaces (for Backward Compatibility)

```
char *strncpy(char *dst, const char *src, size_t len);

void foo() {
    _Array_ptr<char> s1 : count(10) = malloc(10);
    _Array_ptr<char> s2 : count(5) = malloc(5);
    ...
    dst = strncpy(s1, s2, 5);
    ...
}
```



```
error: passing '_Array_ptr<char>' to parameter of incompatible type 'const char *'
    strncpy(s1, s2, 5);
            ^~
```

Bounds-safe Interfaces (for Backward Compatibility)

```
char *strncpy(_Array_ptr<char> dst : count(n),
              _Array_ptr<const char> src : count(n), size_t n);

void foo() {
    _Array_ptr<char> s1 : count(10) = malloc(10);
    _Array_ptr<char> s2 : count(5) = malloc(5);
    ...
    strncpy(s1, s2, 5); legal
    ...
}
```

Can we have an mechanism working for both checked and unchecked code?

Yes, *by bounds-safe interface*

What about calling `strncpy()` *from unchecked C code?*

```
void bar() {
    char *s1 = malloc(10);
    char *s2 = malloc(5);
    ...
    strncpy(s1, s2, 5);
   ...
}  error: argument has unknown bounds, bounds expected because the 1st parameter has bounds
       strncpy(s1, s2, 5);
               ^~
```

Bounds-safe Interface for `strncpy()`

```
1  char *strncpy(char *dst : itype(_Array_ptr<char>) byte_count(n),
2                const char *src : itype(_Array_ptr<const char>) byte_count(n),
3                size_t n) : itype(_Array_ptr<T>) byte_count(n);
```

`itype` (inter-operation type): a special type that can be either checked or unchecked type, depending on the context

1. dst is set to an itype with bounds of n bytes
2. src is set to an itype with bounds of n bytes
3. return value is set to an itype with bounds of n bytes

```
void bar() {
    char *s1 = malloc(10);
    char *s2 = malloc(5);
    ...
    strncpy(s1, s2, 5); ✔  can compile and run, no bounds checks performed
   ...
}
```

Bounds-safe Interface for `strncpy()`

```
1  char *strncpy(char *dst : itype(_Array_ptr<char>) byte_count(n),
2                const char *src : itype(_Array_ptr<const char>) byte_count(n),
3                size_t n) : itype(_Array_ptr<T>) byte_count(n);


void foo() {
    _Array_ptr<char> s1 : count(10) = malloc(10);
    _Array_ptr<char> s2 : count(5) = malloc(5);
    ...
    strncpy(s1, s2, 5);
    ...
}


    strncpy(s1, s2, 6);
                  error: argument does not meet declared bounds for 2nd parameter
                      strncpy(s1, s2, 6);
                                  ^~~
```

s1 and s2 are bounds-checked at the call site.

✔ *can compile and run, as both src and dst are in bounds*

Function Definition with Bounds-safe Interface Prototype

```
char *strncpy_checkedc(char *dst : itype(_Array_ptr<char>) byte_count(len),
                       char *src : itype(_Array_ptr<char>) byte_count(len),
                       size_t len) : itype(_Array_ptr<char>) byte_count(len) {
    _Checked{
        for (size_t i = 0; i < len; i++) {
            dst[i] = src[i];
        }
    }

    return dst;
}
```

bounds checks for `dst[i]` *and* `src[i]` *inserted and then optimized away by compiler.*

invariant: `i < len`

Function Definition with Bounds-safe Interface Prototype

```
char *foo(char *dst : itype(_Array_ptr<char>) byte_count(len),
          char *src : itype(_Array_ptr<char>) byte_count(len),
          size_t len) : itype(_Array_ptr<char>) byte_count(len) {
    int n = ...
    for (size_t i = 0; i < n; i++) {
        dst[i] = src[i]; no bounds checks for dst[i] and src[i]
    }
    return dst;
}
```

```
char *foo(char *dst : itype(_Array_ptr<char>) byte_count(len),
          char *src : itype(_Array_ptr<char>) byte_count(len),
          size_t len) : itype(_Array_ptr<char>) byte_count(len) {
    int n = ...
    _Checked{
        for (size_t i = 0; i < n; i++) {
            dst[i] = src[i];
        }
    }
    return dst;
}
```

bounds checks for `dst[i]` *and* `src[i]` *inserted by compiler, if compiler cannot prove* `n <= len`

One Small Exercise

Open file `sum2array.c` under the sum2array directory, and refactor the C code to Checked C.

Alternative (and simpler) syntax

```
#define ptr _Ptr
#define array_ptr _Array_ptr
#define nt_array_ptr _Nt_array_ptr
#define checked _Checked
```

(And more)

Defined in `stdchecked.h` (included in Checked C compiler's default search paths for header files)

Dynamic Bounds Cast

`dynamic_bounds_cast<T>()`

- Cast an expression to a checked pointer type; dynamic check inserted, if not provable statically.
- Typically used during converting C code to Checked C.

```
char *strncpy(_Array_ptr<char> dst : count(n),
              _Array_ptr<const char> src : count(n), size_t n);
```

What if the size `n` *is greater than* `dst`*'s real length?*

```
void foo(_Array_ptr<char> dst : count(dst_len), size_t dst_len,
         _Array_ptr<char> src : count(src_len), size_t src_len) {
    ...
    strncpy(dst, src, src_len);
}
```

Trying to reset the bounds of `dst` from `dst_len` to `src_len`.

```
error: it is not possible to prove argument meets declared bounds for 1st parameter
    strncpy(dst, src, src_len);
            ^~~
```

Compiler cannot prove `src_len <= dst_len`!

Dynamic Bounds Cast

`dynamic_bounds_cast<T>()`

- Cast an expression to a checked pointer type, with dynamic checks inserted.
- Typically used during converting C code to Checked C.

```
void foo(_Array_ptr<char> dst : count(dst_len), size_t dst_len,
         _Array_ptr<char> src : count(src_len), size_t src_len) {
    ...

    _Array_ptr<char> target : count(src_len) =
        dynamic_bounds_cast<array_ptr<char>>(dst, count(src_len));

    strncpy(target, src, src_len);
}
```

`dst`'s bounds cast to `src_len`.
Check for `src_len` <= `dst_len` inserted by compiler.

Breaktime (10 mins)

Next: Hands on Practice of Writing Checked C Code

The Tiny BigNum Library

`tiny-bignum-c` is a C Library for arbitrary precision, unsigned integers.

Location of code: /tutorial/tiny-bignum/

```
struct bn {
    /* DTYPE: uint32_t */
    DTYPE array[BN_ARRAY_SIZE];
};

void bignum_add(struct bn* a, struct bn* b, struct bn* c);


void bignum_from_string(struct bn* n, char* str, int nbytes);
void bignum_to_string(struct bn* n, char* str, int maxsize);


Example: tests/factorial.c
```

1. The BigNum struct and the Checked array type

The `_Checked[]` keyword declares a new checked array type just as the checked pointer type.

```
int arr _Checked[10];
arr[9] = 10;
                    A valid dereference
arr[10] = 1:
                    error: out-of-bound memory access
```

1. The BigNum struct and the Checked array type

The `_Checked[]` keyword declares a new checked array type just as the checked pointer type.

```
int arr _Checked[10];
arr[9] = 10;
                A valid dereference
arr[10] = 1;
                error: out-of-bound memory access
```

bn.h Line 77:

```
struct bn {
    DTYPE array _Checked[BN_ARRAY_SIZE];
};
```

Enforces spatial memory safety for variable **`array`**

2. Incremental Refactor with Bounds-Safe Interfaces (itype)

```
1  char *strncpy(char *dst : itype(_Array_ptr<char>) byte_count(n),
2                const char *src : itype(_Array_ptr<const char>) byte_count(n),
3                size_t n) : itype(_Array_ptr<T>) byte_count(n);
```

Function declarations can be enhanced with **bounds-safe interfaces** to interact with checked pointers and raw C pointers simultaneously in the same function.

Now lets modify the following functions with itypes.

```
void bignum_assign(struct bn* dst, struct bn* src);           /* bn.h: 118 */
void bignum_dec(struct bn* n);                                /* bn.h: 115 */
void bignum_mul(struct bn* a, struct bn* b, struct bn* c);    /* bn.h:  99 */
void bignum_to_string(struct bn* n, char* str, int maxsize); /* bn.h:  94 */
```

2. Incremental Refactor with Bounds-Safe Interfaces (itype)

```
1  char *strncpy(char *dst : itype(_Array_ptr<char>) byte_count(n),
2                const char *src : itype(_Array_ptr<const char>) byte_count(n),
3                size_t n) : itype(_Array_ptr<T>) byte_count(n);
```

Function declarations can be enhanced with **bounds-safe interfaces** to interact with checked pointers and raw C pointers simultaneously in the same function.

Now lets modify the following functions with itypes.

```
void bignum_assign(struct bn* dst: itype(_Ptr<struct bn>),
                   struct bn* src: itype(_Ptr<struct bn>));
void bignum_dec(struct bn* n: itype(_Ptr<struct bn>));
void bignum_mul(struct bn* a: itype(_Ptr<struct bn>),
                struct bn* b: itype(_Ptr<struct bn>),
                struct bn* c: itype(_Ptr<struct bn>));
void bignum_to_string(struct bn* n: itype(_Ptr<struct bn>),
                      char* str: itype(_Array_ptr<char>) byte_count(maxsize),
                      int maxsize);
```

3. tests/factorial.c: Checked Region

Checked Region can be toggled using

```
_Checked void fn() {
  /* checked scope is entire fn */
}
```

OR

```
_Checked {
  /* block level checked scope */
}
```

3. tests/factorial.c: Checked Region

Checked Region can be toggled using

```
_Checked void fn() {
  /* checked scope is entire fn */
}
```

OR

```
_Checked {
  /* block level checked scope */
}
```

Address-of operator (&) automatically generates check pointers in a Checked Region.

```
_Checked {
  struct foo A;
}
```

→ **&A** *has type* _Ptr<**struct** foo>

Run ./build/test_factorial to see the result.

4. Linked List Sample

Try to refactor `linkedlist/sample.c` on your own!

1. Swap legacy C pointers with Checked Pointers
2. Use modified malloc and free function calls

    `malloc<int>(size)` and `free<int>(ptr)` for an `int` (array) pointer.

3. Identify appropriate checked regions.

Breaktime (30 mins)

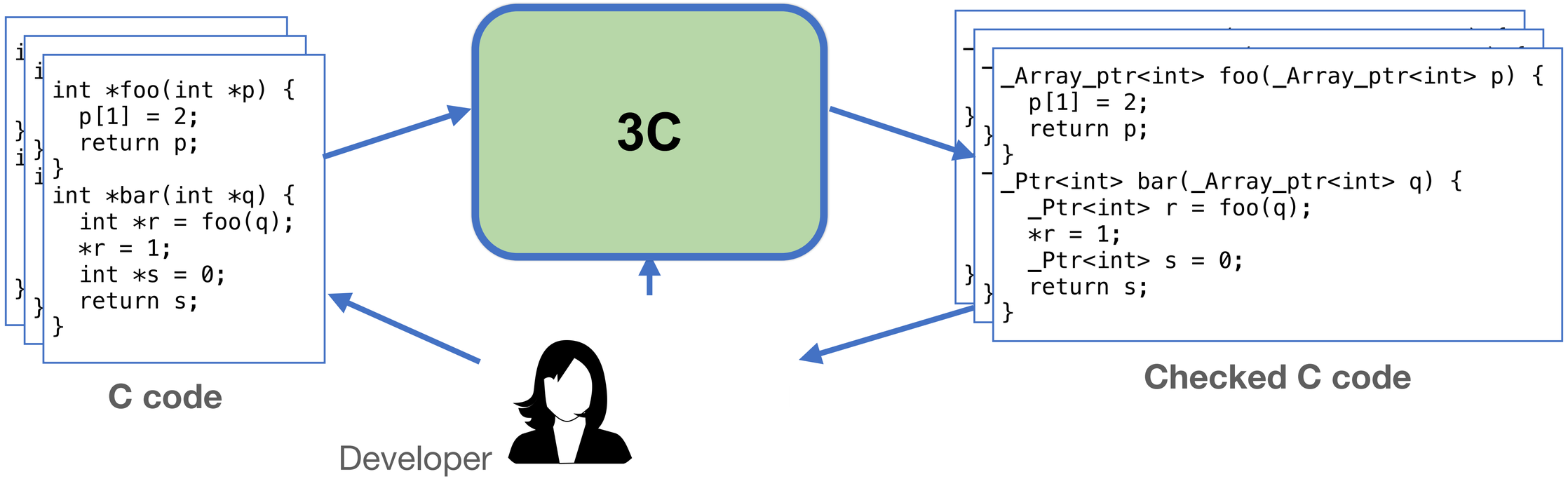
Next: Introducing 3C
Semi-Automatically porting from C to Checked C

Making Legacy C Code Safe

- Many attempts to automatically compile C to something safe
  - CCured, Softbound/CETS, Address Sanitizer (ASAN), …
  - But **too slow**: Requires **fat pointers** and **runtime checks** to automatically handle most C idioms (**50% - 2X slowdown**)
  - Too **inexpressive**: Some idioms can't be supported at all
  - Too **little control**: Unsupported idioms hard to fix, or ignored by tool, making safety uncertain

- The 3C approach: **Assist programmer/expert move to Checked C**
  - Ensures **safety**, **performance**, **maintainability**

3C : Iterative, Automated Conversion

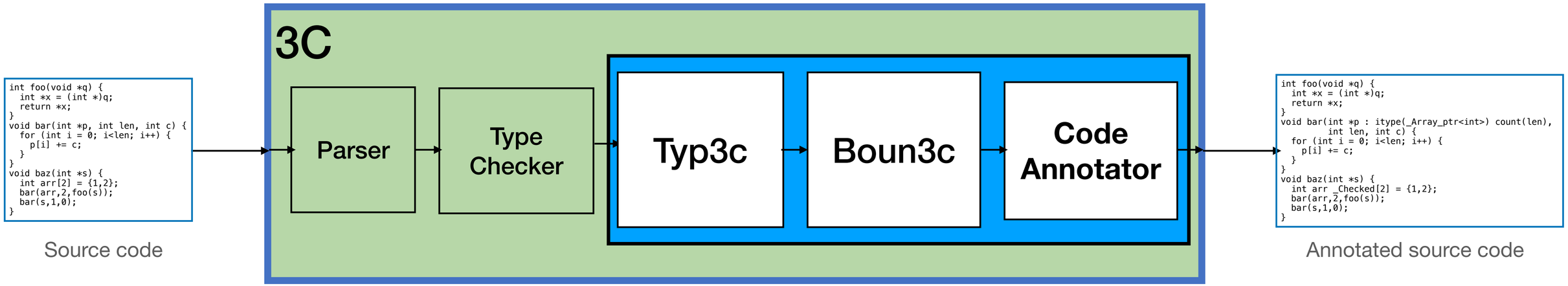


- **3C** is designed to be used with the **developer in the loop**
  - **Annotates automatically**
  - Identifies and **prioritizes root causes** of conversion problems
  - **Modularizes** areas in need of refactoring

3C Algorithm Overview

- Whole program **static analysis** to determine pointer types, array bounds
- Fully automated analysis will never be perfect. 3C aims to **efficiently leverage developer**
  - Localize safe usage
  - Isolate root causes of non-safety

3C Algorithm Overview



- **Typ3c** infers which pointers should have *checked types*
- **Boun3c** infers *bounds* for array-type pointers
- The **annotator** updates the original source code

Details in OOPSLA'22 paper: `https://dl.acm.org/doi/abs/10.1145/3527322`

Typ3c

- **Approach:** Determine, for each pointer:
  - Is used safely (**chk**) or not (**wild**)? How is it used? (**ptr**, **arr**, **ntarr**)
  - *Whole-program, linear-time analysis*
    - Basic approach takes inspiration from CCured, CQual

- Several novel features to accelerate porting
  - **Root cause** analysis
  - **Modular** checked-pointer identification
    - Using bounds-safe interfaces

Boun3c

- Idea: **Allocation sites are *seeds*** for bounds, and pointers ***flow*** **with their bounds** via assignments and calls

```
int x[10];  /* x seeded with count(10) */
int *p = x; /* x's bound flows to p */
foo(p,10);  /* p and bound 10 flow to foo() */
```

- This is a kind of **correlation analysis** (inspiration from Locksmith)
  - Context sensitive for function calls, assignments to data structures
- Use **heuristics** when lacking seed information (e.g., library)

Examples

Original C

```
void foo(int *a) {
  int *b = a;
  *b = 1;
}
void bar(void) {
  int x[10];
  int *p = x;
  foo(p);
  p[2] = 1;
}
```

Converted Checked C

```
void foo(_Ptr<int> a) {
  _Ptr<int> b = a;
  *b = 1;
}
void bar(void) {
  int x _Checked[10];
  _Array_ptr<int> p : count(10) = x;
  foo(p);
  p[2] = 1;
}
```

Original C

```
void remember(void *x);
void foo(int *a) {
  int *b = a;
  remember(b);
  *b = 1;
}
void bar(void) {
  int x[10];
  int *p = x;
  foo(p);
  p[2] = 1;
}
```

Converted Checked C

```
void remember(void *x);
void foo(int *a : itype(_Ptr<int>)) {
  int *b = a;
  remember(b);
  *b = 1;
}
void bar(void) {
  int x _Checked[10];
  _Array_ptr<int> p : count(10) = x;
  foo(p);
  p[2] = 1;
}
```

- remember(…) is a possibly unsafe extern function
- Its usage is isolated within foo using an itype

Original C

```
_Itype_for_any(T) void reme
void foo(int *a) {
  int *b = a;
  remember(b);
  *b = 1;
}
void bar(void) {
  int x[10];
  int *p = x;
  foo(p);
  p[2] = 1;
}
```

Converted Checked C

```
_Itype_for_any(T) void remember(void
void foo(_Ptr<int> a) {
  _Ptr<int> b = a;
  remember<int>(b);
  *b = 1;
}
void bar(void) {
  int x _Checked[10];
  _Array_ptr<int> p : count(10) = x;
  foo(p);
  p[2] = 1;
}
```

- To correct the conversion, we can annotate the original C code
- We could have also corrected the converted code, and rerun it through 3C

Original C

```c
#include <stdlib.h>
void baz(int *q,
         int *c, int len) {
  for (int i = 0; i<len; i++)
    q[i] += *c;
  }
}
void bar(int n) {
  int *p =
    malloc(sizeof(int)*n);
  int c = 5;
  baz(p,&c,n);
}
```

Converted Checked C

```c
#include <stdlib.h>
void baz(_Array_ptr<int> q : count(len),
         _Ptr<int> c, int len) {
  for (int i = 0; i<len; i++) {
    q[i] += *c;
  }
}
void bar(int n) {
  _Array_ptr<int> p : count(n) =
    malloc<int>(sizeof(int)*n);
  int c = 5;
  baz(p,&c,n);
}
```

- The **`malloc`** in **`bar`** acts as a seed bound; flows into **`baz`** with its length
- But **`baz`** would have inferred length via heuristic too, on its own

Full-program Porting Demo

- First taste of porting: the `tiny-bignum-c` library
  - Step by step demo-port available here: https://github.com/secure-sw-dev/checkedc-tiny-bignum-c/

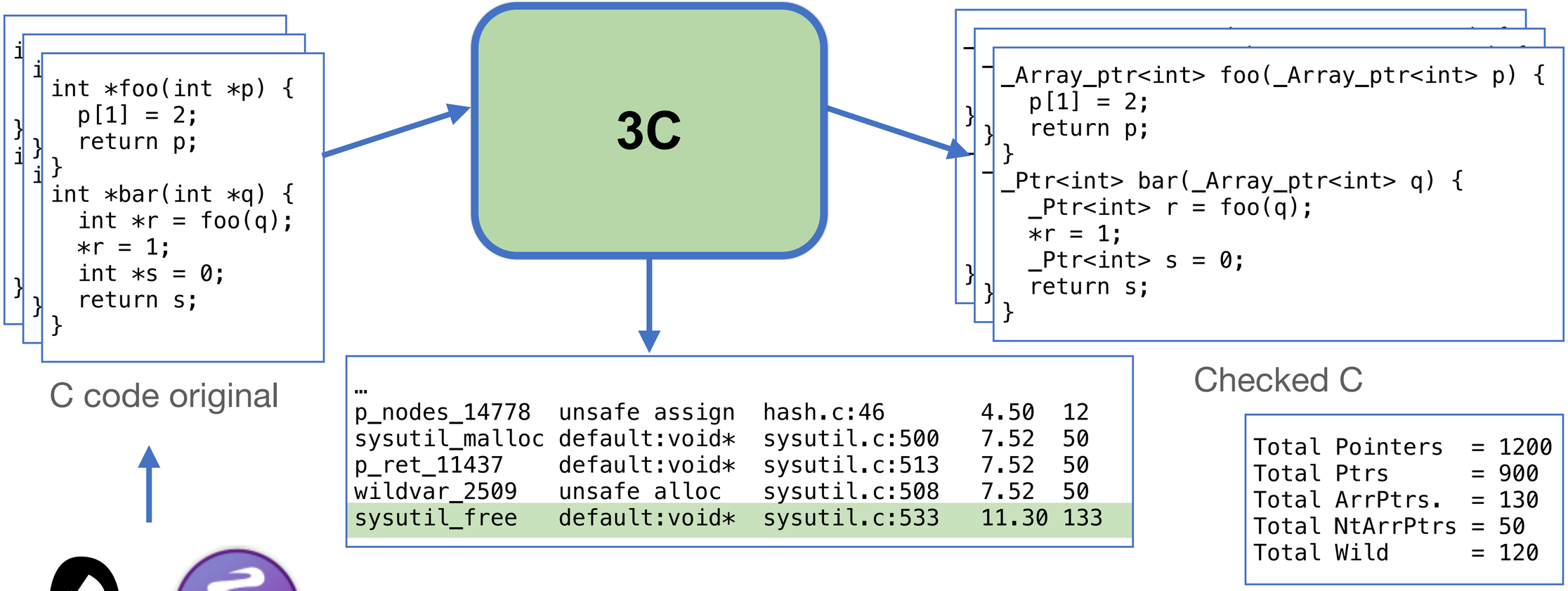
- Go to the 3c/`tiny-bignum-c` directory in the Docker image
  - We will walk through the demo together, following the steps in the README

- What you will learn:
  - Basic use of 3C
  - How using 3C to port to Checked C can reveal vulnerabilities

Iterative Porting

- Most porting experiences are not like tiny-bignum-c
  - Various unsafe, uncheckable, or hard-to-analyze coding idioms prevent 3C from converting portions of the program

- Rather than (only) add more automation fanciness to 3C, we advocate an iterative porting process, in two phases.
  - In both: Human in the loop refactors or corrects code that 3C handles incompletely or incorrectly
  - Aim: 80% machine, 20% human



Phase 1:

- **Run 3C** on the whole project
- Find **root causes** with significant impact
- **Fix the *original .c code*** to address the root cause
  - Might be an annotation, might be a refactoring
- **Repeat**

```
int *foo(int *p) {
  p[1] = 2;
  return p;
}
int *bar(int *q) {
  int *r = foo(q);
  *r = 1;
  int *s = 0;
  return s;
}
```

C code original

3C

```
_Array_ptr<int> foo(_Array_ptr<int> p) {
  p[1] = 2;
  return p;
}
_Ptr<int> bar(_Array_ptr<int> q) {
  _Ptr<int> r = foo(q);
  *r = 1;
  _Ptr<int> s = 0;
  return s;
}
```

Checked C

Phase 2:

- Commit to using Checked C code
- **Fix remaining problems**, file by file
- Use **3C to help propagate changes**
  - To bounds, itypes, etc.

Challenge: Third-party Libraries

- One reason that 3C might not get very far: third-party library use with no Checked C-annotated headers
  - Recall that Checked C ships with many libc headers annotated with itypes, e.g., stdlib.h, stdio.h, string.h, etc.

- Upon discovering this: Use 3C to construct a properly annotated header, based on the client-program's use of the library
  - Then use that header to attempt to convert the client program
  - Port the third-party library at a future time

Iterative Porting Demo

- Next taste of porting: the `libjpeg` client-program demo
  - Step-by-step demo available here: `https://github.com/secure-sw-dev/libjpeg_tutorial`

- Go to the `3c/libjpeg` directory in the Docker image
  - We will walk through the demo together, following the steps in the README

- What you will learn: How the 3C's root-cause analysis directs you to problems to address during phase 1, and fixes to make in phase 2
  - One problem to solve: Annotate libjpeg header

To Try on Your Own

- Port one of the projects at https://github.com/kokke/
  - Designed for embedded devices. Clean code, not too big

- Feeling ambitious? Try porting a benchmark from https://github.com/secure-sw-dev/checkedc/wiki/Benchmarks-for-evaluating-Checked-C
  - We have already ported vsftpd; will make that public soon

- If you completely port anything, let me know!

Where to go next with 3C?

- **Improve Developer Experience**
  - Integrate with clangd, for use in IDE
- **Expand Palette of Analyses and Results**
  - Support inference of bounds(…) annotations, generic types
  - Develop new backends; e.g., to add value to follow-on analysis
- **Lifecycle and Maintenance**
  - Support erasable Checked C syntax, so that residual code is legal C. Can be compiled, analyzed, fuzzed, etc. using C-based tools.
- **Safety of unchecked code**
  - Integration with sandboxing via RLbox (new annotations)