Toward Verification of Unchecked Codes in Checked C

Saeed Nejati
Mentor: David Tarditi

University of Waterloo

Microsoft

December 6th, 2019

Motivation

- Checked C: An extension of C designed for adding memory safety

Checked C

Motivation

- Checked C: An extension of C designed for adding memory safety
- Bounds information for memory regions and pointers

Checked C

Motivation

- Checked C: An extension of C designed for adding memory safety
- Bounds information for memory regions and pointers
  - `_Ptr<int> p`

Checked C

Motivation

- Checked C: An extension of C designed for adding memory safety
- Bounds information for memory regions and pointers
  - `_Ptr<int> p`
  - `_Array_ptr<int> a : bound(a, a+n)`

Checked C

Motivation

- Checked C: An extension of C designed for adding memory safety
- Bounds information for memory regions and pointers
  - _Ptr<int> p
  - _Array_ptr<int> a : bound(a, a+n)
  - _Nt_array_ptr<char> c : count(n)

Checked C

Motivation

- Checked C: An extension of C designed for adding memory safety
- Bounds information for memory regions and pointers
  - `_Ptr<int> p`
  - `_Array_ptr<int> a : bound(a, a+n)`
  - `_Nt_array_ptr<char> c : count(n)`
- Incremental adoption

Checked C

Motivation

- Checked C: An extension of C designed for adding memory safety
- Bounds information for memory regions and pointers
  - `_Ptr<int> p`
  - `_Array_ptr<int> a : bound(a, a+n)`
  - `_Nt_array_ptr<char> c : count(n)`
- Incremental adoption
  - Inter-operation of checked and unchecked codes

C | Checked C

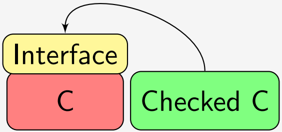
Motivation

- Checked C: An extension of C designed for adding memory safety
- Bounds information for memory regions and pointers
  - _Ptr<int> p
  - _Array_ptr<int> a : bound(a, a+n)
  - _Nt_array_ptr<char> c : count(n)
- Incremental adoption
  - Inter-operation of checked and unchecked codes
- Calling Unchecked functions from Checked

C | Checked C

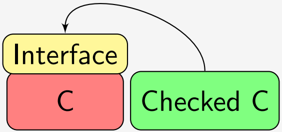
Motivation

- Checked C: An extension of C designed for adding memory safety
- Bounds information for memory regions and pointers
  - `_Ptr<int> p`
  - `_Array_ptr<int> a : bound(a, a+n)`
  - `_Nt_array_ptr<char> c : count(n)`
- Incremental adoption
  - Inter-operation of checked and unchecked codes
- Calling Unchecked functions from Checked
  - Only through Bounds-safe interface



Motivation

- Checked C: An extension of C designed for adding memory safety
- Bounds information for memory regions and pointers
  - `_Ptr<int> p`
  - `_Array_ptr<int> a : bound(a, a+n)`
  - `_Nt_array_ptr<char> c : count(n)`
- Incremental adoption
  - Inter-operation of checked and unchecked codes
- Calling Unchecked functions from Checked
  - Only through Bounds-safe interface
  - e.g. `int *a : count(n)`



Motivation

- Checked C: An extension of C designed for adding memory safety
- Bounds information for memory regions and pointers
  - `_Ptr<int> p`
  - `_Array_ptr<int> a : bound(a, a+n)`
  - `_Nt_array_ptr<char> c : count(n)`
- Incremental adoption
  - Inter-operation of checked and unchecked codes
- Calling Unchecked functions from Checked
  - Only through Bounds-safe interface
  - e.g. `int *a : count(n)`

Question

How do we provide security guarantees for a mix of checked and unchecked C codes?

Motivation

- Pointers to memory regions are passed to an unchecked function

Motivation

- Pointers to memory regions are passed to an unchecked function
- Bounds-safe interface: Partial specification of the boundaries

Motivation

- Pointers to memory regions are passed to an unchecked function
- Bounds-safe interface: Partial specification of the boundaries
- Q: Does the function access those memory regions within their boundaries?

Motivation

- Pointers to memory regions are passed to an unchecked function
- Bounds-safe interface: Partial specification of the boundaries
- Q: Does the function access those memory regions within their boundaries?

Goal

Verify the safety of unchecked functions against their bounds-safe interface.

Outline

Ep. 1: Finding Violations (Fantastic Bugs and Where to Find them)

Clang Static Analyzer

- Checked C is implemented on top of Clang

Clang Static Analyzer

- Checked C is implemented on top of Clang
- Clang has a Static Analyzer

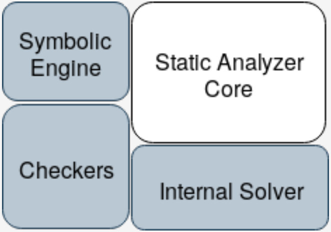
Clang Static Analyzer

- Checked C is implemented on top of Clang
- Clang has a Static Analyzer
- Use it to find memory bugs

Clang Static Analyzer

- Checked C is implemented on top of Clang
- Clang has a Static Analyzer
- Use it to find memory bugs

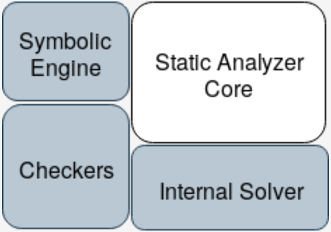
- Core engine
  - Explores all paths
  - Tracks program states
  - Maintains a hierarchical memory model



Clang Static Analyzer

- Checked C is implemented on top of Clang
- Clang has a Static Analyzer
- Use it to find memory bugs

- Core engine
  - Explores all paths
  - Tracks program states
  - Maintains a hierarchical memory model
- A set of checkers that look for specific types of bugs



Clang Static Analyzer

- **Problem 1:** None of the checkers make use of the available Bounds information

Clang Static Analyzer

- Performs Symbolic Execution

```
int x;
int y;
int z = 2 * x;
int w = y - z;
```

```
x : $1
y : $2
z : 2 * $1
w : $2 - (2 * $1)
```

Clang Static Analyzer

- Performs Symbolic Execution

```
int x;                                    x : $1
int y;                                    y : $2
int z = 2 * x;                            z : 2 * $1
int w = y - z;                            w : $2 - (2 * $1)
```

- Internal solver
  - Limited power on handling complex arithmetic
  - Reasons about the ones with concrete starting point

Clang Static Analyzer

- Performs Symbolic Execution

```
int x;                x : $1
int y;                y : $2
int z = 2 * x;        z : 2 * $1
int w = y - z;        w : $2 - (2 * $1)
```

- Internal solver
  - Limited power on handling complex arithmetic
  - Reasons about the ones with concrete starting point
- Bounds-safe information:

```
void foo(int *a: count(n), int n);
```

- **Problem 1:** None of the checkers make use of the available Bounds information.
- **Problem 2:** Bounds are commonly defined over non-concrete symbols.

SimpleBounds Checker

A new checker:

- Reads and make use of Bounds expressions
- Uses SMT solvers for handling complex bounds checking expression

Bounds Checking and Solvers

- Checking if the accessed location is within bounds

Bounds Checking and Solvers

- Checking if the accessed location is within bounds
- $LowerBound \leq Index < UpperBound$

Bounds Checking and Solvers

- Checking if the accessed location is within bounds
- $LowerBound \leq Index < UpperBound$
- Safety question: Is $Index$ always in bounds?

Bounds Checking and Solvers

- Checking if the accessed location is within bounds
- $LowerBound \leq Index < UpperBound$
- Safety question: Is $Index$ always in bounds?
- Query for the negated version:

Bounds Checking and Solvers

- Checking if the accessed location is within bounds
- $LowerBound \leq Index < UpperBound$
- Safety question: Is $Index$ always in bounds?
- Query for the negated version:
  - $(Index < LowerBound) \vee (Index \geq UpperBound)$

Bounds Checking and Solvers

- Checking if the accessed location is within bounds
- $LowerBound \leq Index < UpperBound$
- Safety question: Is $Index$ always in bounds?
- Query for the negated version:
  - $(Index < LowerBound) \vee (Index \geq UpperBound)$
- No solution: There is no index that goes out of bounds!

Bounds Checking and Solvers

- Checking if the accessed location is within bounds
- $LowerBound \leq Index < UpperBound$
- Safety question: Is $Index$ always in bounds?
- Query for the negated version:
  - $(Index < LowerBound) \vee (Index \geq UpperBound)$
- No solution: There is no index that goes out of bounds!
- i.e. It is safe!

Bounds Checking and Solvers

- Checking if the accessed location is within bounds
- $LowerBound \leq Index < UpperBound$
- Safety question: Is $Index$ always in bounds?
- Query for the negated version:
  - $(Index < LowerBound) \vee (Index \geq UpperBound)$
- No solution: There is no index that goes out of bounds!
- i.e. It is safe!
- Solution: working example that could breaks the code!

SimpleBounds Checker

```
void foo(int *p: count(n), int n);
                    // count(n) expands to bounds(p, p+n)
                // Assume the range is not invalid (n > 0)
```

SimpleBounds Checker

```
void foo(int *p: count(n), int n);
                       // count(n) expands to bounds(p, p+n)
                 // Assume the range is not invalid (n > 0)
void foo(int *p, int n) {
```

SimpleBounds Checker

```
void foo(int *p: count(n), int n);
                   // count(n) expands to bounds(p, p+n)
              // Assume the range is not invalid (n > 0)
void foo(int *p, int n) {
    int *a = p;                // Aliasing can be handled
```

SimpleBounds Checker

```
void foo(int *p: count(n), int n);
                  // count(n) expands to bounds(p, p+n)
             // Assume the range is not invalid (n > 0)
void foo(int *p, int n) {
    int *a = p;                    // Aliasing can be handled
    a[n / 2] = 1;
```

$(n/2 < 0) \lor (n/2 \geq n)$

SimpleBounds Checker

```
void foo(int *p: count(n), int n);
                   // count(n) expands to bounds(p, p+n)
              // Assume the range is not invalid (n > 0)
void foo(int *p, int n) {
    int *a = p;                    // Aliasing can be handled
    a[n / 2] = 1;
                                        // this should be ok
```

$(n/2 < 0) \vee (n/2 \geq n)$

SimpleBounds Checker

```
void foo(int *p: count(n), int n);
                  // count(n) expands to bounds(p, p+n)
              // Assume the range is not invalid (n > 0)
void foo(int *p, int n) {
    int *a = p;                   // Aliasing can be handled
    a[n / 2] = 1;
                                       // this should be ok

    int k = n + n;
```

$(n/2 < 0) \vee (n/2 \geq n)$

SimpleBounds Checker

```
void foo(int *p: count(n), int n);
                    // count(n) expands to bounds(p, p+n)
                 // Assume the range is not invalid (n > 0)
void foo(int *p, int n) {
    int *a = p;                    // Aliasing can be handled
    a[n / 2] = 1;
                                     // this should be ok
    int k = n + n;
    a[k] = 1;
```

$(n/2 < 0) \vee (n/2 \geq n)$

$(n + n < 0) \vee (n + n \geq n)$

SimpleBounds Checker

```c
void foo(int *p: count(n), int n);
                  // count(n) expands to bounds(p, p+n)
              // Assume the range is not invalid (n > 0)
void foo(int *p, int n) {
    int *a = p;                   // Aliasing can be handled
    a[n / 2] = 1;
                                       // this should be ok
    int k = n + n;
    a[k] = 1;
                                        // Buffer Overflow!
```

$(n/2 < 0) \vee (n/2 \geq n)$

$(n + n < 0) \vee (n + n \geq n)$

SimpleBounds Checker

```
void foo(int *p: count(n), int n);
                    // count(n) expands to bounds(p, p+n)
               // Assume the range is not invalid (n > 0)
void foo(int *p, int n) {
    int *a = p;                      // Aliasing can be handled
    a[n / 2] = 1;
                                          // this should be ok

    int k = n + n;
    a[k] = 1;
                                        // Buffer Overflow!

    int t = (n & 1) | ((n & 1) ^ 1);
                                        // This is always 1
```

$(n/2 < 0) \vee (n/2 \geq n)$

$(n + n < 0) \vee (n + n \geq n)$

SimpleBounds Checker

```
void foo(int *p: count(n), int n);
                    // count(n) expands to bounds(p, p+n)
                 // Assume the range is not invalid (n > 0)
void foo(int *p, int n) {
    int *a = p;                  // Aliasing can be handled
    a[n / 2] = 1;
                                       // this should be ok
    int k = n + n;
    a[k] = 1;
                                       // Buffer Overflow!
    int t = (n & 1) | ((n & 1) ^ 1);
                                       // This is always 1
    a[t - 2] = 1;
```

$(n/2 < 0) \vee (n/2 \geq n)$

$(n + n < 0) \vee (n + n \geq n)$

SimpleBounds Checker

```
void foo(int *p: count(n), int n);
                    // count(n) expands to bounds(p, p+n)
                 // Assume the range is not invalid (n > 0)
void foo(int *p, int n) {
    int *a = p;                        // Aliasing can be handled
    a[n / 2] = 1;                          (n/2 < 0) ∨ (n/2 ≥ n)
                                             // this should be ok
    int k = n + n;
    a[k] = 1;                              (n + n < 0) ∨ (n + n ≥ n)
                                              // Buffer Overflow!
    int t = (n & 1) | ((n & 1) ^ 1);
                                              // This is always 1
    a[t - 2] = 1;                            // Buffer Underflow!
```

$(n/2 < 0) \vee (n/2 \geq n)$

$(n + n < 0) \vee (n + n \geq n)$

SimpleBounds Checker

```
void foo(int *p: count(n), int n);
                    // count(n) expands to bounds(p, p+n)
               // Assume the range is not invalid (n > 0)
void foo(int *p, int n) {
    int *a = p;                    // Aliasing can be handled
    a[n / 2] = 1;                          (n/2 < 0) ∨ (n/2 ≥ n)
                                          // this should be ok
    int k = n + n;
    a[k] = 1;                            (n + n < 0) ∨ (n + n ≥ n)
                                          // Buffer Overflow!
    int t = (n & 1) | ((n & 1) ^ 1);
                                          // This is always 1
    a[t - 2] = 1;                        // Buffer Underflow!
}
```

SimpleBounds Checker

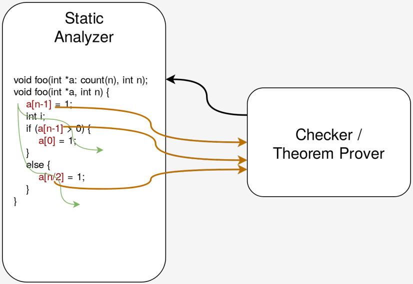
- ArrayBound checkers of clang static analyzer do not detect these (underflow/overflow) bugs

SimpleBounds Checker

- ArrayBound checkers of clang static analyzer do not detect these (underflow/overflow) bugs
- Merged into Master (PR #737)

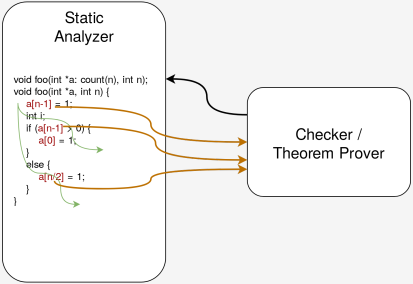
Limitations

- Clang Static Analyzer is as good as its checkers
- Checkers power is bound by the information provided by the core engine
- Very limited power on loops and recursion



Limitations

- Clang Static Analyzer is as good as its checkers
- Checkers power is bound by the information provided by the core engine
- Very limited power on loops and recursion




```
void foo(int *a : count(n), int n);
void foo(int *a : count(n), int n) {
    int i;
    for(i=0; i<n+1; i++)
        a[i] = 0;
}
```

```
void foo(int *a : count(n), int n);
void foo(int *a : count(n), int n) {
    int i;
    for(i=n+1; i>=0; i--)
        a[i] = 0;
}
```

Ep.2: Safety Checking (Fantastic Bugs: The Crimes of Programmer)

Program Safety

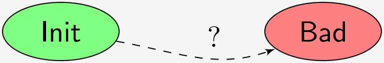
- How checked regions are protected?

Program Safety

- How checked regions are protected?
  - Statically known OOB are caught by compiler
  - Otherwise a dynamic check is inserted to prevent runtime OOB

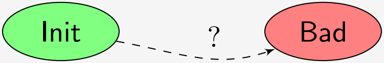
Program Safety

- How checked regions are protected?
  - Statically known OOB are caught by compiler
  - Otherwise a dynamic check is inserted to prevent runtime OOB
- Safety question: Is there an input that makes the program go into a bad state?



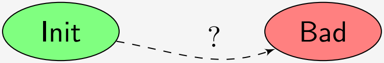
Program Safety

- How checked regions are protected?
  - Statically known OOB are caught by compiler
  - Otherwise a dynamic check is inserted to prevent runtime OOB
- Safety question: Is there an input that makes the program go into a bad state?
- Bad state is being out of bound



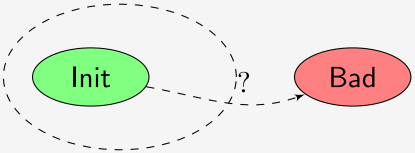
Program Safety

- How checked regions are protected?
  - Statically known OOB are caught by compiler
  - Otherwise a dynamic check is inserted to prevent runtime OOB
- Safety question: Is there an input that makes the program go into a bad state?
- Bad state is being out of bound
- i.e. Will any of the assertions fail?



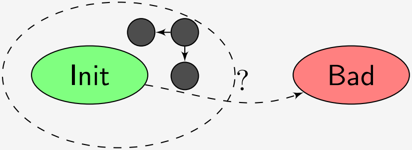
Program Safety

- How checked regions are protected?
  - Statically known OOB are caught by compiler
  - Otherwise a dynamic check is inserted to prevent runtime OOB
- Safety question: Is there an input that makes the program go into a bad state?
- Bad state is being out of bound
- i.e. Will any of the assertions fail?



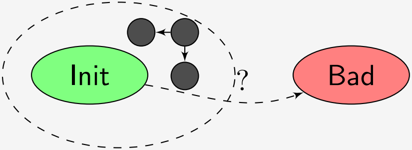
Program Safety

- How checked regions are protected?
  - Statically known OOB are caught by compiler
  - Otherwise a dynamic check is inserted to prevent runtime OOB
- Safety question: Is there an input that makes the program go into a bad state?
- Bad state is being out of bound
- i.e. Will any of the assertions fail?



Program Safety

- How checked regions are protected?
  - Statically known OOB are caught by compiler
  - Otherwise a dynamic check is inserted to prevent runtime OOB
- Safety question: Is there an input that makes the program go into a bad state?
- Bad state is being out of bound
- i.e. Will any of the assertions fail?



$$
\begin{aligned}
Init &\Rightarrow Inv \\
Inv(X) \wedge Tr(X, X') &\Rightarrow Inv(X') \\
Inv &\Rightarrow \neg Bad
\end{aligned}
$$

Seahorn

- Seahorn is a software verification framework

Seahorn

- Seahorn is a software verification framework
- For LLVM-based languages

Seahorn

- Seahorn is a software verification framework
- For LLVM-based languages
- Based on state-of-the-art model checking and abstract interpretation

Seahorn

- Seahorn is a software verification framework
- For LLVM-based languages
- Based on state-of-the-art model checking and abstract interpretation
- Encodes the state transition as Constraint Horn Clauses

Seahorn

- Seahorn is a software verification framework
- For LLVM-based languages
- Based on state-of-the-art model checking and abstract interpretation
- Encodes the state transition as Constraint Horn Clauses

```
{Pre: x_old = x, y_old = y}
int n = non_deterministic_value();
while (n--) {
    int t1 = x;
    int t2 = y;
    x = t1 + 1;
    y = t2 - 1;
}
{Post: x_old + y_old == x + y}
```

Unchecked Code Verification

- **Problem 1:** How do we put verification conditions at each memory access location?

Unchecked Code Verification

- **Problem 1:** How do we put verification conditions at each memory access location?
- Use the dynamic checks as verification conditions

Unchecked Code Verification

- **Problem 1:** How do we put verification conditions at each memory access location?
- Use the dynamic checks as verification conditions
- **Problem 2:** There is no dynamic checks in unchecked codes

Unchecked Code Verification

- **Problem 1:** How do we put verification conditions at each memory access location?
- Use the dynamic checks as verification conditions
- **Problem 2:** There is no dynamic checks in unchecked codes
- Use the same logic as in checked codes to inject checks for unchecked pointers

Unchecked Code Verification

- **Problem 1:** How do we put verification conditions at each memory access location?
- Use the dynamic checks as verification conditions
- **Problem 2:** There is no dynamic checks in unchecked codes
- Use the same logic as in checked codes to inject checks for unchecked pointers
- More relaxed than being a checked code

Unchecked Code Verification

- **Problem 1:** How do we put verification conditions at each memory access location?
- Use the dynamic checks as verification conditions
- **Problem 2:** There is no dynamic checks in unchecked codes
- Use the same logic as in checked codes to inject checks for unchecked pointers
- More relaxed than being a checked code
- Assumption: all function calls within an unchecked region should call to functions with bounds-safe interface

Unchecked Code Verification

Replace the front-end of Seahorn with Checked C clang

Unchecked Code Verification

Replace the front-end of Seahorn with Checked C clang

1. Add bounds for unchecked pointers

Unchecked Code Verification

Replace the front-end of Seahorn with Checked C clang

1. Add bounds for unchecked pointers
2. Inject verification sink functions (`__VERIFIER_error`) in the LLVM bit-code at dynamic check points

Unchecked Code Verification

Replace the front-end of Seahorn with Checked C clang

1. Add bounds for unchecked pointers
2. Inject verification sink functions (`__VERIFIER_error`) in the LLVM bit-code at dynamic check points
3. Query Seahorn back-end for program safety

Unchecked Code Verification

Replace the front-end of Seahorn with Checked C clang

1. Add bounds for unchecked pointers
2. Inject verification sink functions (`__VERIFIER_error`) in the LLVM bit-code at dynamic check points
3. Query Seahorn back-end for program safety

Merged into Master (PR #736)

Unchecked Code Verification

```
int sum(int *a : count(n), int n);
int sum(int *a, int n) {



  int i = 0, s = 0;
  for(i=0; i<n; i++) {
    s += a[i+1];


  }

  a[n / 2] = 1;

  return s;
}
```

Unchecked Code Verification

```
int sum(int *a : count(n), int n);
int sum(int *a, int n) {
  assume(a != NULL);
  assume(n > 0);

  int i = 0, s = 0;
  for(i=0; i<n; i++) {
    s += a[i+1];


  }

  a[n / 2] = 1;

  return s;
}
```

Unchecked Code Verification

```
int sum(int *a : count(n), int n);
int sum(int *a, int n) {
  assume(a != NULL);
  assume(n > 0);

  int i = 0, s = 0;
  for(i=0; i<n; i++) {
    s += a[i+1];
    sassert(i+1 < n);
    sassert(i+1 >= 0);
  }

  a[n / 2] = 1;

  return s;
}
```

Unchecked Code Verification

```
int sum(int *a : count(n), int n);
int sum(int *a, int n) {
  assume(a != NULL);
  assume(n > 0);

  int i = 0, s = 0;
  for(i=0; i<n; i++) {
    s += a[i+1];
    sassert(i+1 < n);
    sassert(i+1 >= 0);
  }

  a[n / 2] = 1;

  return s;
}
```

The result should be "SAT": there exists a path to failing an assertion!

Limitations

- Checked C+Seahorn is sound if the conditions are sound
- Conditions are sound if the bounds inference is sound
- Seahorn works completely on LLVM side.

Conclusions

Demo.

Summary

Having an unchecked function with bounds-safe interface:

- Bounds-aware static analyzer checker finding OOB accesses
- Safety verification with Checked C+Seahorn

- Identifying limitations of static analysis and verification in Checked C
- Paths for future work in verification of checked/unchecked C codes

Future Work

- Bounds for unchecked pointers:
  - back-propagating the assumptions
  - forward-propagating the bounds
- Improving bounds inference
- Expanding the checker

Thanks!

Questions?