FINDING ITERATOR-RELATED ERRORS WITH CLANG STATIC ANALYZER

Ádám Balogh
Ericsson Hungary Ltd.
adam.balogh@ericsson.com

Euro LLVM 2018
CLANG STATIC ANALYZER

- Symbolic execution of the program to find errors
- Path sensitive walk on the Control Flow Graph
- Simulated execution
  - On every possible path
  - Variables are represented as symbolic values
- Constraints are calculated for symbolic values for each path
- Possible paths are calculated based on the constraints
- Checkers may store additional data for variables in every state
- Checkers may spawn new execution paths or terminate existing ones
#include <stdlib.h>
void test(int b)
{
    int a, c;
    switch (b){
    case 1:  a = b / 0; break;
    case 4:
        c = b-4;
        a = b / c; break;
    }
}
ITERATORS IN C++

› Types that can be used to identify and traverse the elements of a container
› No common ancestor like in Java, C# or Objective-C
› Minimal set of common properties:
  – Copy-constructible, copy-assignable, destructible
  – Can be incremented: operator++(), both prefix and postfix
  – Can be dereferenced: operator*()
› Different categories: input, output, forward, bi-directional, random-access
› **Difficulty for static analysis: how to recognize a type as an iterator?**
DANGERS OF ITERATORS

› Dereferencing an iterator outside of its range
  – Typically dereferencing the past end iterator
› Access of an invalidated iterator
› Mismatch between container and iterator or two iterators
› All these errors lead to undefined behavior which is hard to debug
› Surprisingly, Static Analyzer could not find any of these errors until now
OUT-OF-RANGE DEREFERENCING AN ITERATOR

Simple Example 1
```cpp
auto i = v.end();
*i; // Oops!
```

Simple Example 2
```cpp
auto i = v.begin();
*--i; // Oops!
```

Typical Example
```cpp
auto first = std::find(V.begin(), V.end(), e);
auto &x = *first; // What if e is not found in V?
```
INVALIDATED ITERATOR ACCESS

Simple Example

```cpp
auto i0 = L.begin();
L.erase(i0);
*i0;
```

Typical Example

```cpp
for (auto i = L.begin(); i != L.end(); ++i) {
    if (dislike(*i))
        L.erase(i);
}
```
ITERATOR MISMATCH

Typical Example 1

```cpp
data first = std::find(V1.begin(), V1.end(), e);
V2.erase(first); // Undefined behavior
```

Typical Example 2

```cpp
data first = std::find(V1.begin(), V1.end(), e);
if (first == V2.end()) // Always false!!
   return;
auto &x = *first;
```
OUR SOLUTION

› Combined checker for all three kinds of errors
› Checks for the different kinds of errors can be enabled separately
› Based on STL containers, but also works for custom container types with certain STL like properties:
   – `std::list`-like containers: no subscript operator
   – `std::vector`-like containers: subscript operator and only back-modifiable
   – `std::deque`-like containers: subscript operator and also front-modifiable (e.g. `push_front()`)
› We regard types with `iterator` or `iter` as the suffix in their name as iterators if they fulfill the set of minimal requirements for iterators
EXAMPLE: PAST-END ACCESS

```cpp
#include <vector>
int test(std::vector<int> v, int n) {
  auto i = std::find(v.begin(), v.end(), n);
  return *i;
}
```
#include <vector>

int test(std::vector<int> v, int n) {
    auto i = std::find(v.begin(), v.end(), n);
    if (i == v.end())
        return 0;
    return *i;
}
Modelling Iterators

- F was conjured (synthetized) for a return value of a function, e.g. std::find()
- No order known between B, E and F unless assumed in a branch
HANDLING BEGIN AND END

› Begin and end position symbols of a container are initially undefined
› A symbol is conjured upon first call to `begin()` or `end()` as the iterator’s position
   – Later calls return the same symbol for the iterator’s position
› The `begin()` and `end()` symbols are removed from the container data when they become invalidated
   – New value assigned to the container
   – Container’s `clear()` method is called
   – Data is moved from the container using `std::move()`
Methods modifying the container (not its elements) also affect iterator positions

Upon insertions and deletions:
- All iterator positions of the container are checked
- Some of them are invalidated (according to the standard)
- The rest is shifted to match the new arrangement
- If the inserted or deleted position is relative to the begin or end of the container:
  - Shift the begin or end of the container as well
EXAMPLE: INSERT INTO VECTOR
EXAMPLE: INSERT INTO VECTOR
EXAMPLE: INSERT INTO VECTOR

Note: other positions (e.g. F and F+1) are not invalidated if we do not know whether they are indeed after the insertion
- If we already have such assumption in the current state, we invalidate them as well
EXAMPLE: PUSH FRONT INTO LIST

Container:

Iterators:

Begin: \( B \)
End: \( E \)
EXAMPLE: PUSH FRONT INTO LIST
HANDLING MOVE SEMANTICS

› Standard: upon move constructor or move assignment, the existing iterators remain valid, but refer to the element in the new container, **except the past-end iterators**

› Upon move:
  - Move the begin-symbol to the new container
  - Reassign all iterators to the new container
  - Create a new end-symbol for the new container
  - Replace the end-symbol in the reassigned symbolic expressions
EXAMPLE: MOVE SEMANTICS

Container:

Begin: \(B\)
End: \(E\)

Iterators:

F-6
B+7
B+8
E-5
E-5
F
F+1
E-1
E
EXAMPLE: MOVE SEMANTICS

New Container:

Iterators:

Begin: B
End: E’
EXAMPLE: MOVE SEMANTICS

Container:

Begin:  B  
End:    E

Iterators:

F-6  B+7  B+8  E-5  E-5  F  F+1  E  

Finding Iterator-related Errors with Clang Static Analyzer | Euro LLVM 2018, Bristol | © Ericsson Hungary Ltd. | 2018-03-28 | Page 23 (32)
EXAMPLE: MOVE SEMANTICS

Old Container:

Iterators:
Symbolic execution of search algorithms (e.g. std::find()) is too complex
- The analyzer usually cannot determine whether the element is found
- Lots of different execution paths generated

Simplification: we model the search functions of the STL
- Only two paths: the element is found or not
- When found, we conjure a new symbol
- When not found, we return the argument which stores the end of the range

False not-found states?
- Yes, but (as we mentioned it), full execution does not help
- The programmer can use assertions if the element is surely to be found
\textbf{INFRASTRUCTURE LIMITATIONS}

› Tracking of complex structures
  – Static Analyzer can track Symbols or Memory Regions
  – Complex structures may appear as any of them

› Static Analyzer’s default constraint manager only handles integer ranges
  – Microsoft’s Z3 is an option, but it increases analysis time by more than a whole magnitude!
  – We need to compare iterator positions which are symbolic expressions
  › Example: compare iterator positions to past-end iterator position
  › Symbolic expressions: symbol plus/minus constant is enough
  › Record the relation of two positions in the state and use it for subsequent assumptions
Assign iterator positions for both symbols and regions behaving as iterators
Track every assignment in the checker and copy the state manually
Hooks: after constructors, upon value bindings, after temporary creation
There is no hook after temporary creation
  – Analyzer extended by such hook
  – Useful for every future checker that tracks complex structures
We have an expression $M + a @ N + b$

- $M$ and $N$ are symbols, $a$ and $b$ concrete integers and $@$ is a comparison operator

Rearrange it to $M - N @ b - a$

- Constraint manager can store an integer range for $M - N$ now

What about overflow cases?

- Type extension disables correct handling of intentional overflow cases
- Solution: only do the rearrangement if $M$, $N$, $a$ and $b$ are signed and inside $(\text{MIN}/4..\text{MAX}/4)$
- This limitation is acceptable not only for the iterator checkers, but also other checkers, e.g. array out-of-range checkers

Side effect: also do the rearrangement if $@$ is an additive operator

- No limitation in this case
If we store a range for $M - N$ in the constraint manager, it still cannot reason about $N - M$.

Solution: if constraint manager cannot find a range for $N - M$, then try to find it for $M - N$ and then negate the range as well.

It can later be extended to a more generic solution for other negation cases.
CURRENT STATUS

› Checker is under review on the Phabricator in 10 parts
  – First part is accepted
  – Some other parts tentatively accepted (dependent on yet unaccepted parts)
› Infrastructure patches (except difference negation) already in Clang
› Whole checker is internally used inside Ericsson
OPEN ISSUES

Problems causing these false-positives:
- Container’s length() is not simulated
- Random-access iterators are not specially handled
- Difficult to determine whether two containers are indeed different
CONCLUSION

› New checker developed to detect the 3 most typical error using iterators
› Clang Static Analyzer core infrastructure improved
› Existing checkers benefit from core infrastructure improvements
› New checkers may be developed based on these improvements
THANKS

› Artem Dergachev
› Devin Coughlin
› George Karpenkov
› Gábor Horváth