Static Analysis of Reference-Counted Objects for the C Programming Language

2025-10-13

Ole Wiedemann, Volkmar Sieh

Friedrich-Alexander-Universität Erlangen-Nürnberg
This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 539710462.

informatik 4
systemsoftware

Friedrich-Alexander-Universität
Faculty of Engineering

DFG

Motivation

- The C programming language is widely used in systems programming

Motivation

- The C programming language is widely used in systems programming
  - ⚠ Memory safety remains an issue

Motivation

- The C programming language is widely used in systems programming
  - ⚠ Memory safety remains an issue
- External tooling and programming techniques to increase safety

Motivation

- The C programming language is widely used in systems programming
  - ⚠ Memory safety remains an issue
- External tooling and programming techniques to increase safety
- Reference counting to track allocations

Motivation

- The C programming language is widely used in systems programming
  - ⚠ Memory safety remains an issue
- External tooling and programming techniques to increase safety
- Reference counting to track allocations
  - ⚠ Usage can be error-prone

Motivation

- The C programming language is widely used in systems programming
  - ⚠ Memory safety remains an issue
- External tooling and programming techniques to increase safety
- Reference counting to track allocations
  - ⚠ Usage can be error-prone

→ Static analysis to check reference counting

Motivation

- The C programming language is widely used in systems programming
  - ⚠ Memory safety remains an issue
- External tooling and programming techniques to increase safety
- Reference counting to track allocations
  - ⚠ Usage can be error-prone

→ Static analysis to check reference counting
  - ⚠ Rely on conventions or heuristics

Motivation

- The C programming language is widely used in systems programming
  - ⚠ Memory safety remains an issue
- External tooling and programming techniques to increase safety
- Reference counting to track allocations
  - ⚠ Usage can be error-prone

→ Static analysis to check reference counting
  - ⚠ Rely on conventions or heuristics
  - ⚠ Perform inter-procedural analysis

Motivation

- The C programming language is widely used in systems programming
  - ⚠ Memory safety remains an issue
- External tooling and programming techniques to increase safety
- Reference counting to track allocations
  - ⚠ Usage can be error-prone

→ Static analysis to check reference counting

- ⚠ Rely on conventions or heuristics
- ⚠ Perform inter-procedural analysis
- ⚠ Limited support for language features

The Problem

Reference Counting

```
1 ref_init(x);    // Initializes the counter to 1
2 ref_acquire(x); // Increments the counter by 1
3 ref_release(x); // Decrements the counter by 1
```

Reference Counting

```
1 ref_init(x);    // Initializes the counter to 1
2 ref_acquire(x); // Increments the counter by 1
3 ref_release(x); // Decrements the counter by 1
```

- Counter is handled by the programmer

Reference Counting

```
1 ref_init(x);    // Initializes the counter to 1
2 ref_acquire(x); // Increments the counter by 1
3 ref_release(x); // Decrements the counter by 1
```

- Counter is handled by the programmer
- Counter must equal the number of references

Reference Counting

```
1 ref_init(x);    // Initializes the counter to 1
2 ref_acquire(x); // Increments the counter by 1
3 ref_release(x); // Decrements the counter by 1
```

- Counter is handled by the programmer
- Counter must equal the number of references
- Incorrect counters can cause memory leaks or use-after-free anomalies

Example

```
1  struct inode *inode_alloc(struct fs *fs) {
2      struct inode *i = ref_alloc(sizeof(*i));
3      if (i == NULL) return NULL;
4      i->i_fs = fs;
5      return i;
6  }
```

Example

```
1 struct inode *inode_alloc(struct fs *fs) {
2     struct inode *i = ref_alloc(sizeof(*i));
3     if (i == NULL) return NULL;
4     i->i_fs = fs;
5     return i;
6 }
```

Q1 Which structures are reference-counted?

Example

```
1  struct inode *inode_alloc(struct fs *fs) {
2      struct inode *i = ref_alloc(sizeof(*i));
3      if (i == NULL) return NULL;
4      i->i_fs = fs;
5      return i;
6  }
```

Q1 Which structures are reference-counted?

Q2 How does `ref_alloc` behave?

Example

```
1  struct inode *inode_alloc(struct fs *fs) {
2      struct inode *i = ref_alloc(sizeof(*i));
3      if (i == NULL) return NULL;
4      i->i_fs = fs;
5      return i;
6  }
```

Q1 Which structures are reference-counted?

Q2 How does `ref_alloc` behave?

Q3 How should `inode_alloc` behave?

Example

```
1  struct inode *inode_alloc(struct fs *fs) {
2      struct inode *i = ref_alloc(sizeof(*i));
3      if (i == NULL) return NULL;
4      i->i_fs = fs;
5      return i;
6  }
```

Q1 Which structures are reference-counted?

Q2 How does `ref_alloc` behave?

Q3 How should `inode_alloc` behave?

Q4 Is this code correct?

Our Approach

Overview

1. **Annotations**
2. **Static analysis**
   - 2.1 Simplification
   - 2.2 Annotation checking
   - 2.3 **State calculation**
   - 2.4 **State checking**

Annotations

```
1 struct inode *inode_alloc(struct fs *fs) {
2     struct inode *i = ref_alloc(sizeof(*i));
3     if (i == NULL) return NULL;
4     i->i_fs = fs;
5     return i;
6 }
```

Q1 Which structures are reference-counted?

Q2 How does `ref_alloc` behave?

Q3 How should `inode_alloc` behave?

Which structures are reference-counted?

```
1 struct inode {
2     ...
3 } REF_TYPE;
```

```
1 struct fs {
2     ...
3 } REF_TYPE;
```

Annotations

```
1 struct inode *inode_alloc(struct fs *fs) {
2     struct inode *i = ref_alloc(sizeof(*i));
3     if (i == NULL) return NULL;
4     i->i_fs = fs;
5     return i;
6 }
```

Q1 Which structures are reference-counted? ✔

Q2 How does `ref_alloc` behave?

Q3 How should `inode_alloc` behave?

How does `ref_alloc` behave?

```
1 extern void REF_TYPE *ref_alloc(size_t size)
2     REF_ACQUIRE_IF(__return__ != NULL, __return__);
```

Annotations

```
1 struct inode *inode_alloc(struct fs *fs) {
2     struct inode *i = ref_alloc(sizeof(*i));
3     if (i == NULL) return NULL;
4     i->i_fs = fs;
5     return i;
6 }
```

Q1 Which structures are reference-counted? ✔

Q2 How does `ref_alloc` behave? ✔

Q3 How should `inode_alloc` behave?

How should `inode_alloc` behave?

```
1  struct inode *inode_alloc(struct fs *fs)
2      REF_ACQUIRE_IF(__return__ != NULL, __return__)
3  {
4      ...
5  }
```

Annotations

```
1  struct inode *inode_alloc(struct fs *fs)
2      REF_ACQUIRE_IF(__return__ != NULL, __return__) {
3      struct inode *i = ref_alloc(sizeof(*i));
4      if (i == NULL) return NULL;
5      i->i_fs = fs;
6      return i;
7  }
```

Q1 Which structures are reference-counted? ✔

Q2 How does `ref_alloc` behave? ✔

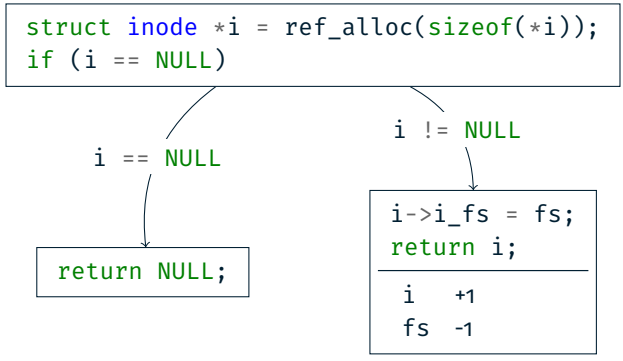
Q3 How should `inode_alloc` behave? ✔

Q4 Is this code correct? ?

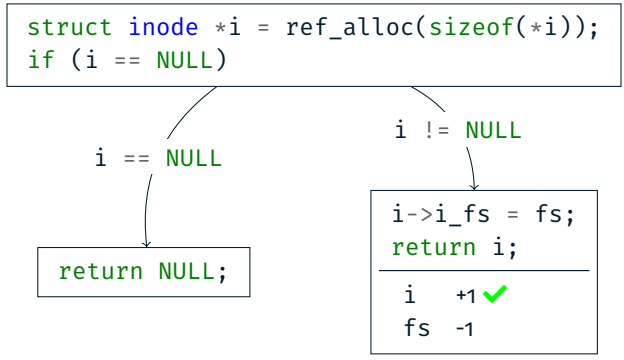
Static Analysis

```
foreach module-ast:
    m = simplify(module-ast)
    check_annotations(m)
    c = control_flow_graph(m)
→   s = calculate_state(c)
→   check_state(m, s)
```

Is this code correct?

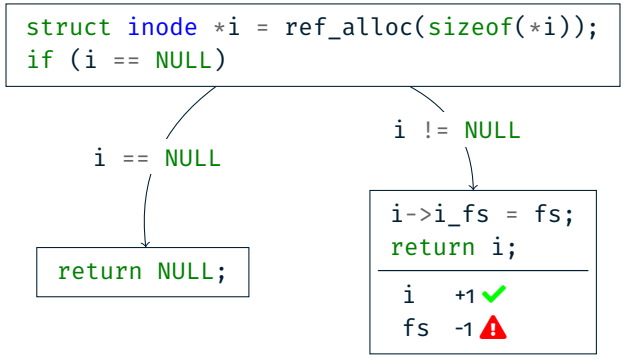


Is this code correct?



- Return value must have a reference delta of +1
  - `REF_ACQUIRE_IF(__return__ != NULL, __return__)`

Is this code correct?



- Return value must have a reference delta of +1
  - `REF_ACQUIRE_IF(__return__ != NULL, __return__)`
- Any other value must have a reference delta of 0

Discussion

Evaluation

- Implemented in the FAUCCC static analyzer

Evaluation

- Implemented in the FAUCCC static analyzer
- Checks the Linux-compatible JITTY operating system

Evaluation

- Implemented in the FAUCCC static analyzer
- Checks the Linux-compatible JITTY operating system
- Catches bugs before they are committed

Evaluation

- Implemented in the FAUCCC static analyzer
- Checks the Linux-compatible JITTY operating system
- Catches bugs before they are committed
  - Makes effectiveness evaluation difficult

Evaluation

- Implemented in the FAUCCC static analyzer
- Checks the Linux-compatible JITTY operating system
- Catches bugs before they are committed
  - Makes effectiveness evaluation difficult
- Number of annotations:
  - 33 / 605 structures ($\sim$ 5%)
  - 220 / 4433 functions ($\sim$ 5%)

Evaluation

- Implemented in the FAUCCC static analyzer
- Checks the Linux-compatible JITTY operating system
- Catches bugs before they are committed
  - Makes effectiveness evaluation difficult
- Number of annotations:
  - 33 / 605 structures ($\sim$ 5%)
  - 220 / 4433 functions ($\sim$ 5%)
- Very fast (<1 second) for most modules

Limitations

- Aliased parameters can cause false negatives

Limitations

- Aliased parameters can cause false negatives
- Reference cycles are not handled

Limitations

- Aliased parameters can cause false negatives
- Reference cycles are not handled
- Conditional reference counting operations are rejected

```
if (a == 2) ref_acquire(x);
/* ... */
if (b == 2) ref_release(x);
```

Limitations

- Aliased parameters can cause false negatives
- Reference cycles are not handled
- Conditional reference counting operations are rejected

```
if (a == 2) ref_acquire(x);
/* ... */
if (b == 2) ref_release(x);
```

→ These limitations are solvable

Future Work

- Apply the approach to other projects (Linux, FreeBSD, ...)

Future Work

- Apply the approach to other projects (Linux, FreeBSD, ...)
- Add initialization analysis

Future Work

- Apply the approach to other projects (Linux, FreeBSD, ...)
- Add initialization analysis
- Add finalization analysis

Conclusion

- Static analyzer for reference-counting
- Annotations instead of conventions or heuristics
- Performs fast intra-procedural analysis

Q & A

For further questions:
wiedemann@cs.fau.de