ParameterGuard

Developing an Advanced Linter and Static Analyzer for Go Programs

Hyunsoo Shin (Lake)

KlaytnFoundation
Developer
lake.shin@klaytn.foundation

September 19, 2023

Table of Contents

Background

Why Golang?

Ease of Learning and Rapid Adoption

Extensive Ecosystem

Maturity and Top-tier Ranking as a Programming Language

Why Golang?

Ease of Learning and Rapid Adoption

Extensive Ecosystem

Maturity and Top-tier Ranking as a Programming Language

Why Golang?

Ease of Learning and Rapid Adoption

Extensive Ecosystem

Maturity and Top-tier Ranking as a Programming Language

Golang's Share in the Blockchain Industry

Ethereum, BNB, Klaytn, Cosmos, Hyperledger Fabric, etc

You Might Encounter Already This Pattern (1/2)

Golang

```go
// []byte type is nilable
func process(serialized []byte) {
    // May cause a panic if `serialize` is empty (nil)
    head := serialized[0]
    ...
    ...
}
```

C

```c
// unsigned char* is nullable
void process(unsigned char* serialized) {
    // (1) May cause a segfault if `serialize` is empty (null)
    // (2) May get a garbage value if `serialize` is empty
    unsigned char head = serialized[0];
    ...
    ...
}
```

You Might Encounter Already This Pattern (2/2)

Golang

```go
func process(serialized []byte) {
    // length check is also legal (len(serialized) != 0)
    if serialized != nil { // Guard for `serialized`
        // Confirmed the `serialized` is not empty
        head := serialized[0]
        ...
    }
}
```

C

```c
void process(unsigned char* serialized) {
    if (serialized != NULL) { // Guard for `serialized`
        // Confirmed the `serialized` is not empty
        unsigned char head = serialized[0];
        ...
    }
}
```

Off-the-shelf Linter and Analyzer

- govet
  - Used to detect and report common code issues, such as variable shadowing, unreachable code, and other potential mistakes
- errcheck
  - Identify and report instances where error values are not properly handled or checked
- ParameterGuard (Work of this presentation)
  - Detecting unsafe usage of function parameters based on heuristic approach

Listed approaches you can take to find a bug

- Fuzzing
  - Pros: If the fuzzing found a bug, that is real bug
  - Cons: Unsound
- Symbolic Execution (Concolic Execution)
  - Pros: Try to explore uncovered paths (coverage)
  - Cons: Partially incomplete and unsound
- Program Verification
  - Pros: Formally verify that a given program contains a bug or not under the well-formed properties
  - Cons: Resource intensive and complexity
- Static Analysis
  - Pros: Sound
  - Cons: Incomplete

Introduction to Static Analysis and its Application

Motivation

Static analysis can effectively detect all potential nil-dereference and nil-access instances without any occurrences of false negatives.

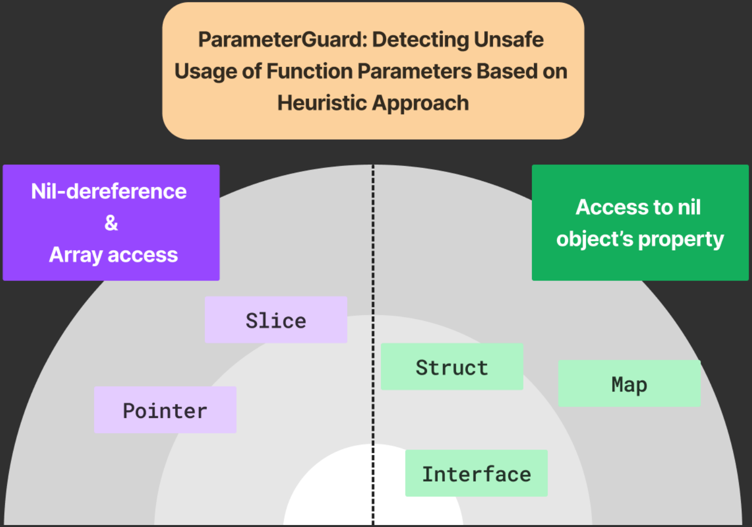
```
func safeProcess(serialized []byte) {
    if serialized != nil { // Guard for `serialized`
        head := serialized[0] // Safe
        ...
    }
}

func unsafeProcess(serialized []byte) {
    head := serialized[0] // Unsafe (To be reported to programmer)
    ...
}
```

Specification & Goal

- Unsafe definition
  - Nil-dereference & Array access
    - Considered types: *Slice* and *Pointer*
  - Access to property of nilable object
    - Considered types: *Struct*, *Map*, and *Interface*
- Efficient Scalability: Complete analysis within 1 minute for all Golang projects
- Clear Explanations: Reports provide violation location and feasible callpath
- Flexible Configuration: e.g., Option to skip analyzing certain packages or functions

Project ParameterGuard



Program Analysis

Static Analysis

In computer science, static program analysis (or static analysis) is the analysis of computer programs performed without executing them, in contrast with dynamic program analysis, which is performed on programs during their execution

By Wikipedia

Bug Taxonomy

Goal

ParameterGuard scrutinizes properties to verify the safe usage of all function parameters.

First Guard Check: Binary Expression

```
if param != nil { ... }
if nil != param { ... }
if len(param) != 0 { ... }
if 0 != len(param) { ... }
```

Second Guard Check: Type switch statement

```
switch v := param.(type) {
    case Object: ...
    ...
}
```

Example

- Safe usage (Guard1 guarantees that *foo* is not nil)

```
if a.b.ptr != nil { // Guard1
    ptr.myfunc()
}
```

- Unsafe usage (No guard found for the usage of *myfunc*)

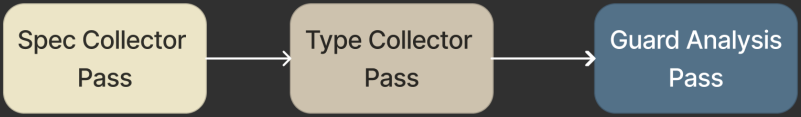
```
a.b.ptr.myfunc()
```

- False positive

```
obj := Obj {
    b: B {
        ptr: func() { ... }
    }
}
call(obj)
```

Implementation

- ParameterGuard was implemented based on Golang analysis package (framework)
- The pass consists of three passes in total
  - Spec Collector Pass
    - Collect all struct type name per package
  - Type Collector Pass
    - Create type mapping table
  - Guard Analysis Pass
    - Inspect an appropriate guard positioned



For more details, visit to project repository
https://github.com/hyunsooda/ParameterGuard

Contribution

- ParameterGuard has identified three crashes caused by nil-dereferences within the Klaytn project
- ParameterGuard has detected a potential nil-dereference issue in yet another open-source project. (https://github.com/fatih/color/pull/203)

Conclusion

- ParameterGuard was implemented in Golang with 1K LoC. https://github.com/hyunsooda/ParameterGuard
- ParameterGuard is the first static analyzer for potential nil-dereference and nil-access pattern detection
- ParameterGuard stands out as a lightweight static analysis tool, boasting the capability to analyze entire Golang projects in approximately 30 seconds
- ParameterGuard incorporates a crucial step, requiring the programmer's final confirmation to filter out false positives