Detection of undefined behavior using KLEE

Pavel Iatchenii
What is undefined behavior?

- **Not** unspecified behavior
- **Not** implementation-defined behavior
- "Behavior, upon the use of a non-portable or erroneous program construct or of erroneous data, for which International Standard imposes no requirements" (C99 standard)
- “Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly...” (comp.lang.c)
UB in symbolic execution

- Injection of checks by KLEE
  - Division by zero
  - Overshift overflow

- Natural processing by KLEE
  - Dereferencing a nullptr
  - Reaching an unreachable program point

- Cases that are hard to catch without code instrumentation
  - Integer overflow
  - Load of invalid enum value
  - Use of a misaligned pointer
Motivating examples

Signed integer overflow

*Undefined Behavior*

```c
int abs(int x) {
    if (x <= 0) {
        return -x;
    }
    return x;
}
```

Implicit conversion with data loss

*Unintentional behavior*

```c
unsigned char convert(signed int x){
    // some optional code
    return x;
}
```
UndefinedBehaviorSanitizer (LLVM)

- *Code generator*, uses compile-time instrumentation to insert certain checks along with handlers

- *Runtime*, implements those handlers and exits the program if so configured
UndefinedBehaviorSanitizer (KLEE)

- LLVM code generator as is, adding `-fsanitize=` flags to Clang compiler is sufficient to instrument bitcode

- **Adopted** LLVM runtime, to accurately analyse the passed arguments containing source location, kind of check, and values
Default compilation

```c
int abs(int x) {
    if (x <= 0) {
        return -x;
    }
    return x;
}
```

6:

```
%7 = load i32, i32* %3, align 4
%8 = sub nsw i32 0, %7
store i32 %8, i32* %2, align 4
br label %11
```
Compilation using UBSan

```c
int abs(int x) {
    if (x <= 0) {
        return -x;
    }
    return x;
}
```

6:

```llvm
%7 = load i32, i32* %3, align 4
%8 = call { i32, i1 } @llvm.ssub.with.overflow.i32 (i32 0, i32 %7)
%9 = extractvalue { i32, i1 } %8, 0
%10 = extractvalue { i32, i1 } %8, 1
%11 = xor i1 %10, true
br i1 %11, label %14, label %12
```

12:

```llvm
%13 = zext i32 %7 to i64, !nosanitize !5
call void @__ubsan_handle_negate_overflow (...)
br label %14
```
Runtime adoption

Common diagnostic emission

```c
static void handleNegateOverflowImpl(
    OverflowData *Data,
    ValueHandle /*OldVal*/) {
    bool IsSigned =
        Data->Type.isSignedIntegerTy();
    ErrorType ET =
        IsSigned ?
            SignedIntegerOverflow :
            UnsignedIntegerOverflow;
    report_error_type(ET);
}
```

Handler

```c
extern "C" void __ubsan_handle_negate_overflow(
    OverflowData *Data,
    ValueHandle OldVal) {
    handleNegateOverflowImpl(
        Data, OldVal
    );
}
```
Results

KLEE extension as an effort to get better in the detection of **undefined behavior** and **unintentional issues**

- The relevant test cases from LLVM sources resulted in error tests, both for **symbolic** and **concrete** values
- Actively used in **UTBotCpp** and succeeds in finding issues on critical projects
References

- Pull request "Support UBSan-enabled binaries" to KLEE mainline
  
  [github.com/klee/klee/pull/1378](github.com/klee/klee/pull/1378)

- UndefinedBehaviorSanitizer documentation
  
  [clang.llvm.org/docs/UndefinedBehaviorSanitizer.html](clang.llvm.org/docs/UndefinedBehaviorSanitizer.html)

- UTBotCpp documentation
  
  [utbot.org](utbot.org)