Análisis Estático con clang-llvm

Marcelo Arroyo
Universidad Nacional de Río Cuarto

$\lambda C.C$ FCEIA - UNR - 2014

Contenidos

Herramientas de desarrollo de software modernos

Objetivos

- Encontrar errores en código fuente.
- Incorporar herramientas automáticas en herramientas de desarrollo.
  - Plugins de compiladores (gcc, clang-llvm, . . .).
  - IDEs.
- Sensibilización del uso de estas herramientas en la industria.

Herramientas de desarrollo de software modernos

Objetivos

- Encontrar errores en código fuente.
- Incorporar herramientas automáticas en herramientas de desarrollo.
  - Plugins de compiladores (gcc, clang-llvm, . . .).
  - IDEs.
- Sensibilización del uso de estas herramientas en la industria.

Desafíos técnicos

- Lograr herramientas de utilidad para el desarrollador.
  - Evitar falsos positivos.
  - Tiempos de ejecución: no mucho mayores a la compilación.
- Embeber componentes (plugins) en compiladores existentes.

Análisis estático de código fuente

- Basados en algún esquema de interpretación abstracta (ej:ejecución simbólica)
- Utilizable en entornos y equipos de desarrollo
- No requiere conocimientos sobre métodos formales
- Herramientas *push and do*
- Quiénes usan estas cosas?
  - OpenSSL (`http://wiki.openssl.org/index.php/Static_and_Dynamic_Analysis`)
  - Google: Thread Safety Analysis (y otros proyectos)
  - ...

Complementos a compiladores

- Transformaciones de código (ej:*mocking*, . . .)
- Generación de stubs (ej: stubs CORBA).
- Análisis de código fuente
  - Estilos de programación.
  - Seguridad (tainted data, . . .)
  - Detección de anti-patterns.
  - Estimación de uso de recursos.
  - Extracción de modelos.
  - Finding bugs.
  - ???

Low Level Virtual Machine (LLVM)

Colección de componentes modulares reusables de compilación (toolchain)

- Soporta compilación estática y dinámica de cualquier lenguaje
- Compilador adoptado por Apple
- Componentes (algunos proyectos relacionados)
  - LLVM core: optimizador y generador de código
  - Clang: C/C++/Objective C front-end
  - vmkit: Implementación de Java y .Net VMs sobre LLVM
  - Klee: ejecución simbólica sobre LLVM para encontrar errores. Puede generar casos de tests.
  - lld: linker clang/llvm

Clang Static Analyzer

- Finding bugs tool (C, C++, Objective C)
- Extensible (ckeckers).
- Es parte del code-base de clang (llvm frontend).
- Soporta *anotaciones* (pragmas y gcc __attribute__())
- Basado en ejecución simbólica (path-sensitive walk of CFG).
- Preciso para encontrar bugs del tipo:
  - Resource leaks
  - Use-after-free
  - En realidad: implementar *type-state systems*

Veamos un ejemplo

Programa C

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, const char * argv[])
{
    char *str1;
    char *str2 = "Hi";

    if (argc > 1)
        str1 = malloc(sizeof(char)*51);

    printf("Hello, World: %s!\n",str1);
    return 0;

}
```

Clang Static Analyzer in Xcode

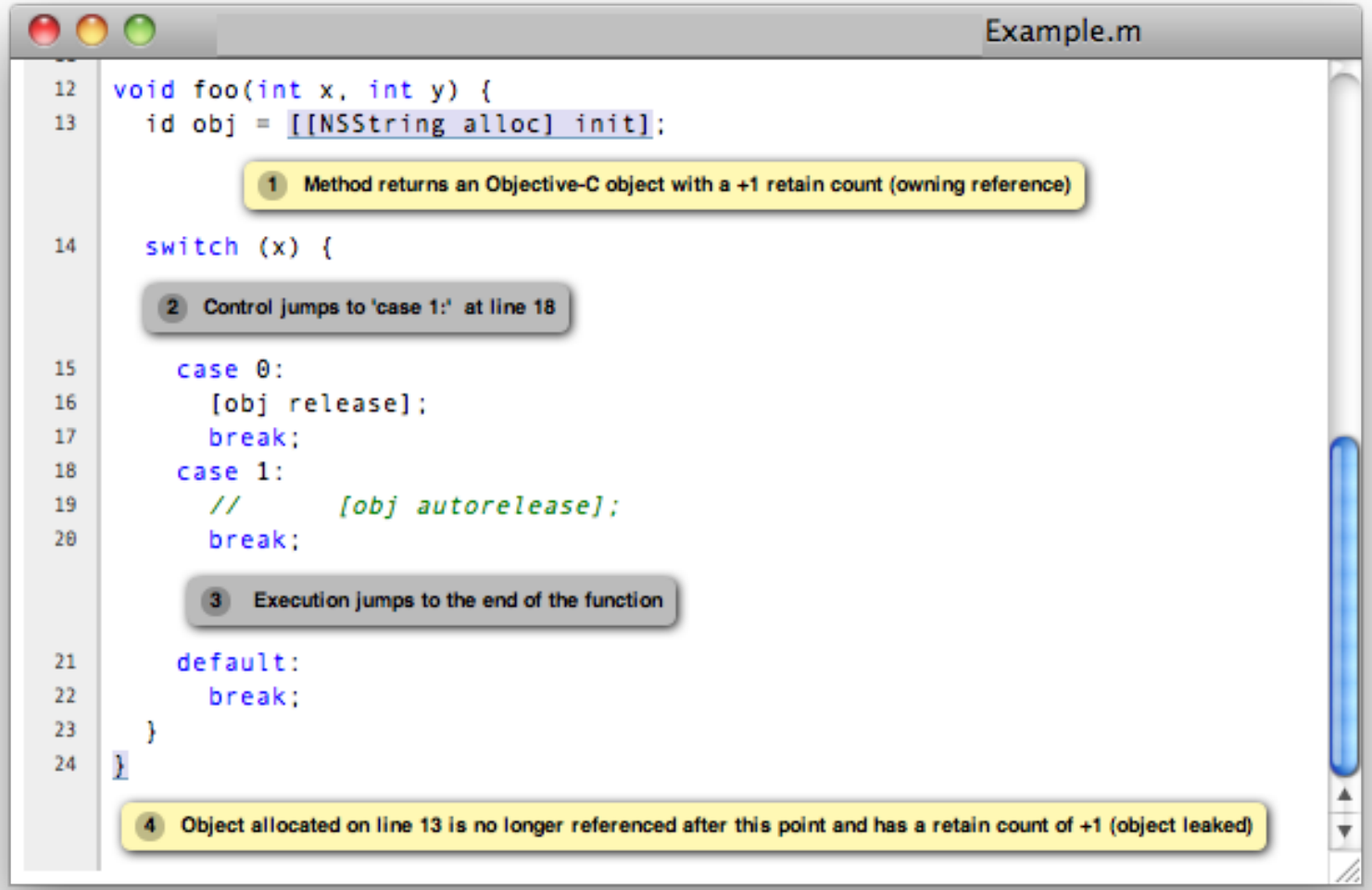
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, const char * argv[])
{
    char *str1;
    char *str2 = "Hi";                                   Unused variable 'str2'

    if (argc > 1)
        str1 = malloc(sizeof(char)*51);

    printf("Hello, World: %s!\n",str1);                  Function call argument is an uninitialized value
    return 0;                                            Potential leak of memory pointed to by 'str1'
}
```

Clang Static Analyzer

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, const char * argv[])
{
    char *str1;
    char *str2 = "Hi";

    if (argc > 1)                                   1. Assuming 'argc' is > 1
        str1 = malloc(sizeof(char)*51);             2. Memory is allocated

    printf("Hello, World: %s!\n",str1);
    return 0;                                       3. Potential leak of memory pointed to by 'str1'
}
```

clang-analyzer warnings (en Xcode)

Clang Static Analyzer



Type-State Systems

Definición: Type-State System (de un tipo T)

$TS_T = (S, O, T, start, err)$

- S es el conjunto de *estados abstractos* del tipo T
- O es el conjunto de *operaciones* sobre objetos de tipo T
- $T : S \times O \rightarrow S$ son las *transiciones* de estados.
- *start* es el estado inicial
- *err* es el estado de error: $T(err, o) = err, \forall o \in O$

Type-State Systems

Definición: Type-State System (de un tipo T)

$TS_T = (S, O, T, start, err)$

- S es el conjunto de *estados abstractos* del tipo T
- O es el conjunto de *operaciones* sobre objetos de tipo T
- $T : S \times O \rightarrow S$ son las *transiciones* de estados.
- *start* es el estado inicial
- *err* es el estado de error: $T(err, o) = err, \forall o \in O$

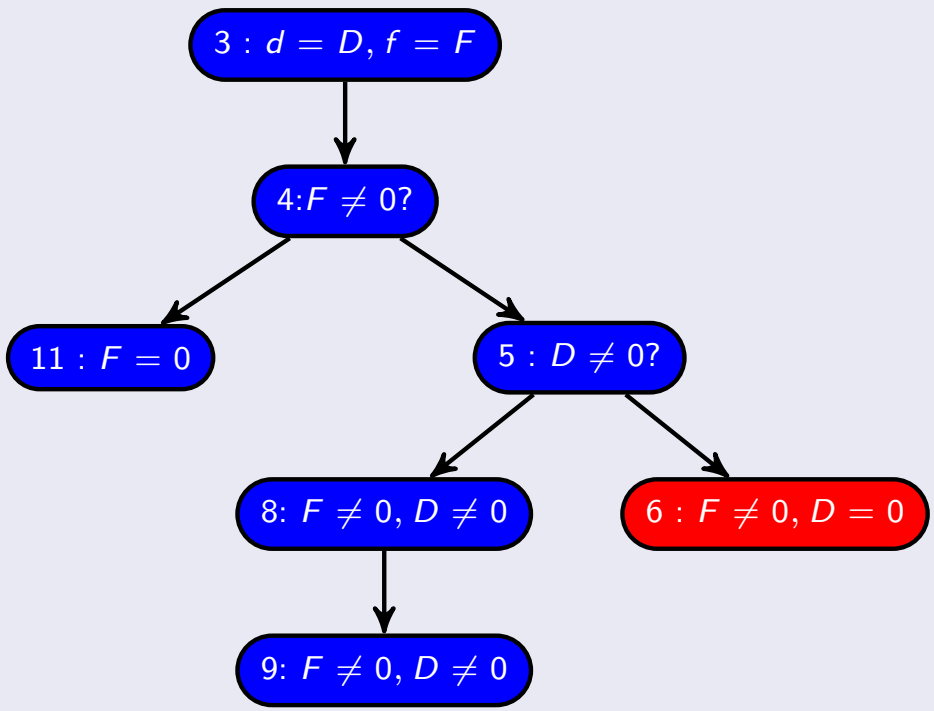
Verificación basado en type-state systems

- Generación del autómata finito en base a cada camino (path)
- Análisis de alcanzabilidad
- Alcanzar un nodo *err* = Camino posible con un error

Clang Analyzer Core

Graph of Reachable Program States

```
1  void WriteLog(char *d) {
2    FILE* f=fopen("app.log","a");
3
4    if ( f != NULL ) {
5      if ( !d )
6        return;
7
8      fputs(d,f);
9      fclose(f);
10    }
11 }
```



Error node

Bugs ∼ Graph reachability

¿Qué contiene un nodo del CFG?

Program Point

- Execution location
- Pre-stmt, post-stmt
- Entering/returning to/from a call
- Stack frame, ...

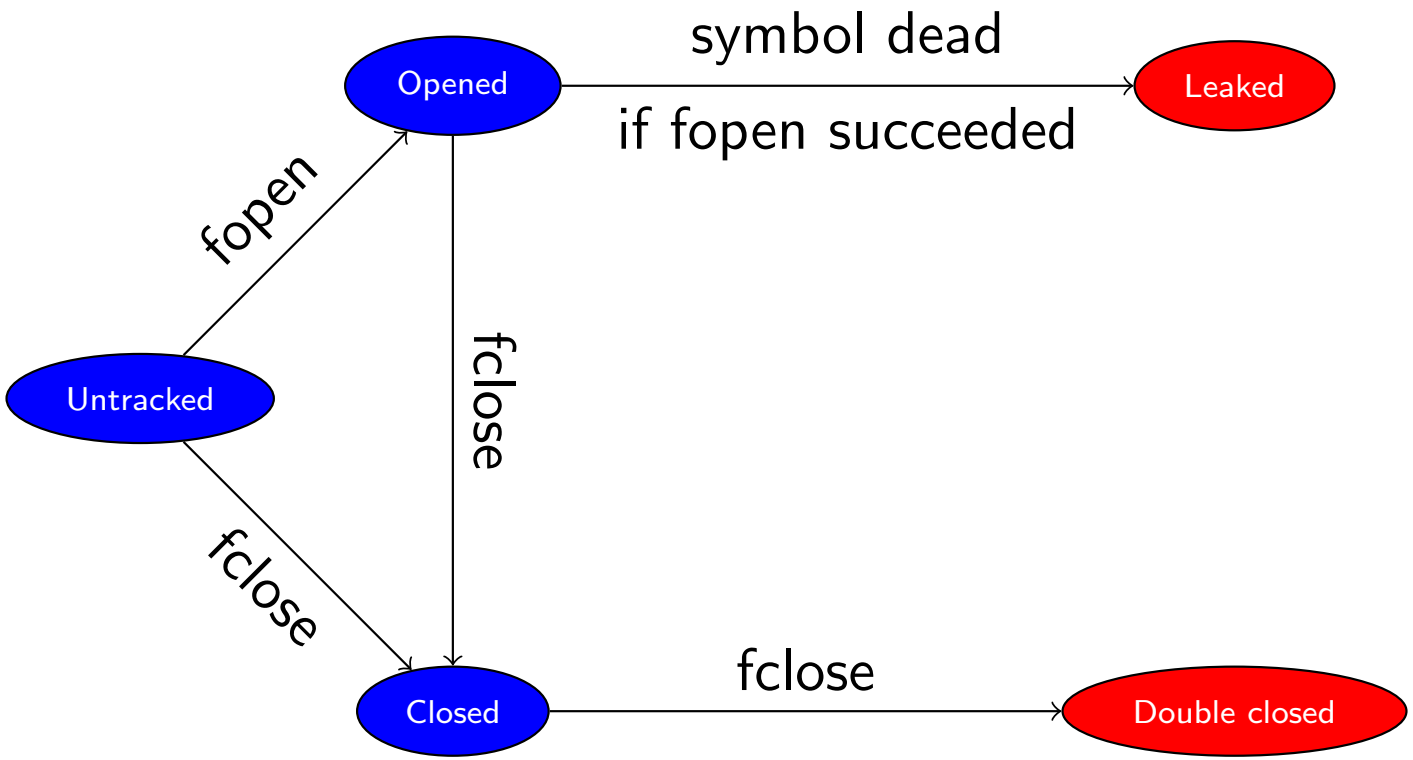
Program State

- Environment: Expr $\rightarrow$ values
- Store: memory locations $\rightarrow$ values
- Constrains on symbolix values
- Generic Data Map (GDM)

Los checkers extienden el estado del CFG

- Crean nuevos nodos de estado.
- Pueden detener la exploración (creando *sink nodes*)
- Los checkers son *visitors*. Ejemplos:
  - `checkPreStmt(const ReturnStmt* stmt, CheckerContext & ctx) const`
    Invocado antes que se entrar a una sentencia return.
  - `checkPostCall(const Callevent& call, CheckerContext & ctx) const`
    Invocado luego de una llamada a función o método.
  - `checkBind(SVal l, Sval r, const Stmt* s, CheckerContext & ctx) const`
    Invocado en cada ligadura de una variable luego de procesar `s`.
- Los checkers son *stateless*.

Un ejemplo: Stream checker



Simple Stream Checker example

Eventos de interés

- Luego de `f = fopen(...)`, agregar transición a *Opened*.
- Antes de la invocación a un `fclose(f)`: Doubleclose error si `f` estaba cerrado. Sino agregar transición a *Closed*.
- Cuando símbolos (ej: `f`) quedan *out of scope*. Reportar bug si estaba *Opened*.
- Cuando el puntero *se escapa* (analyzer no puede seguir su pista). Esto permite eliminar algunos falsos positivos.

Simple Stream Checker example

```
class SimpleStreamChecker : public Checker<check::PostCall,
                                           check::DeadSymbols,
                                           check::PointerEscape> {
public:

  // process fopen()
  void checkPostCall(const CallEvent &Call, CheckerContext &C) const;

  // process fclose()
  void checkPreCall(const CallEvent &Call, CheckerContext &C) const;

  void checkDeadSymbols(SymbolReaper &SR, CheckerContext &C) const;

  ProgramStateRef checkPointerEscape(ProgramStateRef State,
                                     const InvalidatedSymbols &Escaped,
                                     const CallEvent *Call,
                                     PointerEscapeKind Kind) const;

private:
  enum state {Opened, Closed} st;
  REGISTER_MAP_WITH_PROGRAMSTATE(StreamMap, SymbolRef, state)
};
```

Simple Stream Checker example

```
void SimpleStreamChecker::checkPostCall(const CallEvent &Call, CheckerContext &C)
{
  if (!Call.isGlobalFunction("fopen")) return;
  SymbolRef FileDesc = Call.getReturnValue().getAsSymbol();
  ProgramStateRef State = C.getState();
  State = State->set<StreamMap>(FileDesc,Opened);
  C.addTransition(state);
};

void SimpleStreamChecker::checkPreCall(const CallEvent &Call, CheckerContext &C)
{
  if (!Call.isGlobalFunction("fclose") && Call.getNumArgs() != 1) return;
  SymbolRef FileDesc = Call.getArgSVal(0).getAsSymbol();

  // Check double close
  const StreamState *SS = C.getState()->get<StreamMap>(FileDesc);
  if (SS && *SS == closed)
    reportDoubleClose(FileDesc,Call,C);

  ProgramStateRef State = C.getState();
  State = State->set<StreamMap>(FileDesc,Closed);
  C.addTransition(state);
};
```

Algunas ideas de desarrollo

Checkers y herramientas

- Checker para detectar race-conditions y deadlocks in C++ threads.
- Desarrollo de una herramienta de alto nivel (posiblemente gráfica) para generar código de checkers en base a una especificación de *type and state systems*.

Algunas ideas de desarrollo

Checkers y herramientas

- Checker para detectar race-conditions y deadlocks in C++ threads.
- Desarrollo de una herramienta de alto nivel (posiblemente gráfica) para generar código de checkers en base a una especificación de *type and state systems*.

Referencias

- Clang analyzer: `http://clang-analyzer.llvm.org/`
- Zhongxing Xu, Ted Kremenek and Jian Zhang. *A Memory Model for Static Analysis of C Programs*.
- Anna Zacks, Jordan Rose. *How to Write a Checker in 24 Hours*. LLVM Developer's meeting. 2012.

Preguntas?

Gracias...
¿Preguntas?