EXERCISE #18

CFI REVIEW

Write your name and answer the following on a piece of paper

What is the difference between the CFI protections implemented by LLVM and Microsoft's Control Flow Guard?

ADMINISTRIVIA AND ANNOUNCEMENTS

OBJECT-ORIENTED CALL GRAPHS

EECS 677: Software Security Evaluation

Drew Davidson

CLASS PROGRESS

ANALYSIS UNDERLYING OUR
ENFORCEMENT NEEDS

LAST TIME: CFI

REVIEW: LAST LECTURE

LIMIT THE DESTINATION OF INDIRECT CONTROL TRANSFER

Motivation

- Important against return-oriented programming
- Useful against other types of control-hijacking memory attacks as well

Implementation

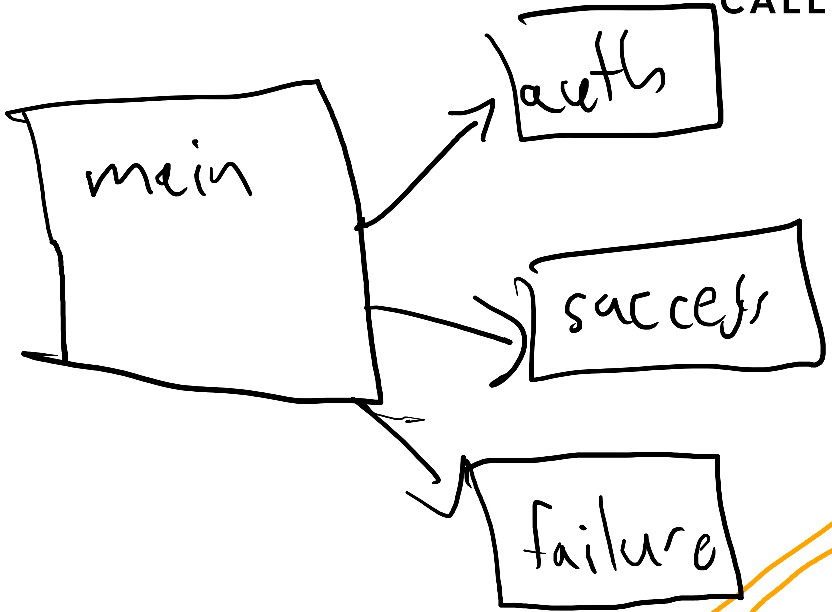
- Requires (over)approximate knowledge of control transfer targets
- Interpose on a control transfer

OVERVIEW

HOW DO WE FIGURE OUT CONTROL TRANSFER TARGETS IN THE FIRST PLACE?

RECALL OUR CFI MOTIVATING EXAMPLE

CALL GRAPH ANALYSIS




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

struct auth {
        char pass[4];
        void (*func)();
};

void success() { printf("Success!\n"); }
void failure() { printf("Failure\n");  }

void auth(struct auth *a) {
        if (strcmp(a->pass, "pass") == 0)
                a->func = &success;
}

int main(int argc, char **argv) {
        struct auth a;
        a.func = failure;

        printf("Enter your password:\n");
        scanf("%s", &a.pass);
        auth(&a);

        a.func();
}
```

LECTURE OUTLINE

- Call Graphs
- Dynamic Dispatch
- Algorithms

CALL GRAPHS

A HISTORY OF COMPUTING

DIRECTED GRAPH OF FUNCTIONS

Simple Form:

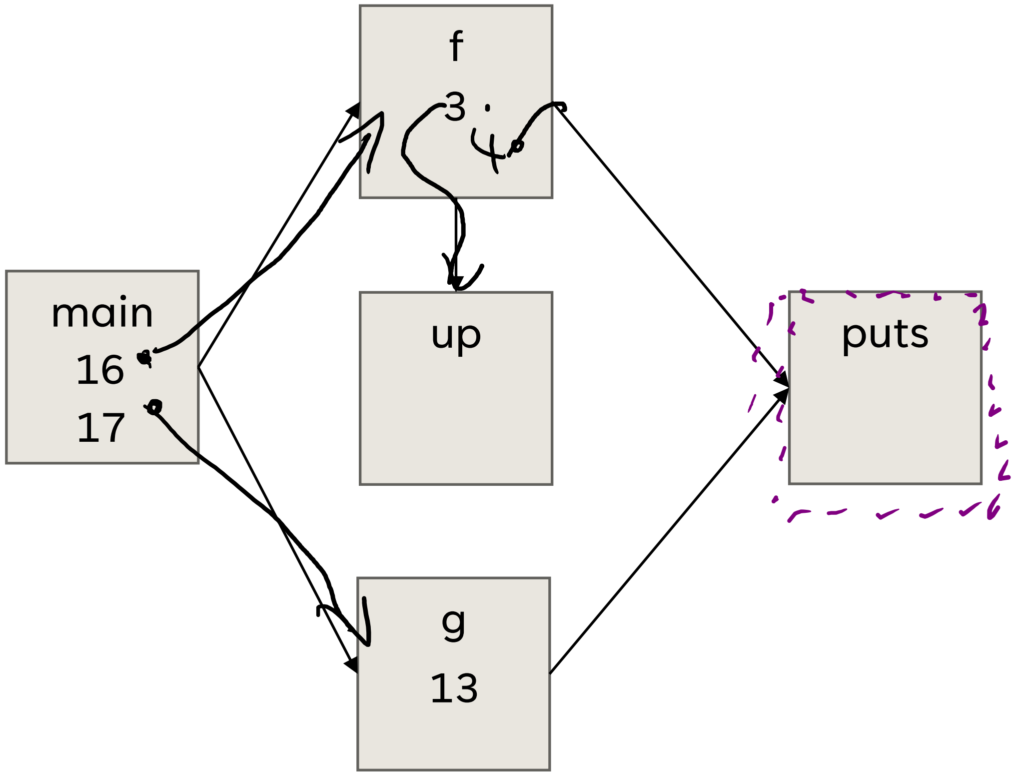
- Node: function

- Edge: function call

Refined Form:

- Node: call site with function "block"

- Edge: function call



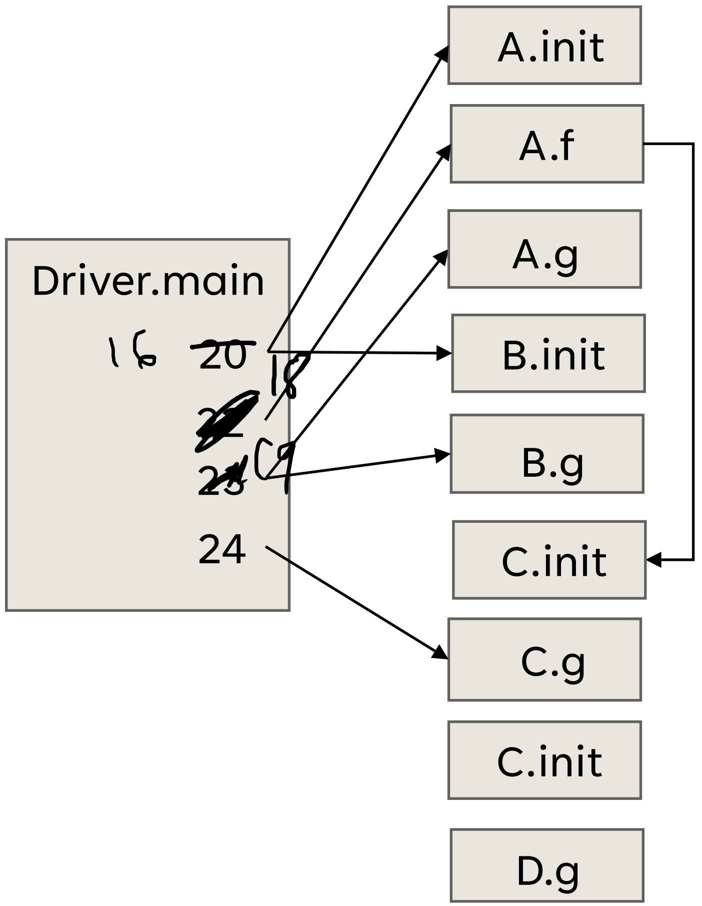

```
1 void f(char * s){
2   for (char *p = s; *p; p++)
3     *p = up(*p);
4   puts(s);
5 }
6
7 char up(char c) {
8   if (c >= 'a' && c <= 'z')
9     return c - ('a' - 'A');
10   return c;
11 }
12
13 void g(){ puts("Bye!"); }
14
15 int main(int c, char *v) {
16   if (argc > 1) { f(v[0]); }
17   g();
18   return 0;
19 }
```

LECTURE OUTLINE

- Call Graphs
- Dynamic Dispatch
- Algorithms

DYNAMIC DISPATCH

A HISTORY OF COMPUTING




```
 1 class A {
 2   public A f(){ return new C(); }
 3   public String g(){ return "A"; }
 4 }
 5 class B extends A{
 6   public String g(){ return "B"; }
 7 }
 8 class C extends A {
 9   public String g(){ return "C"; }
10 }
11 class D extends A {
12   public String g(){ return "D"; }
13 }
14 class Driver {
15   public void main(String[] args){
16     A[] aArr = { new A(), new B()};
17     for (A a : aArr){
18       A res = a.f();
19       print(a.g());
20       print(res.g());
21     }
22   }
23 }
```

DYNAMIC DISPATCH: GETS COMPLICATED!

A HISTORY OF COMPUTING

DYNAMIC DISPATCH: GETS COMPLICATED!

A HISTORY OF COMPUTING

Direct Calls

Not so bad

Indirect Calls

Quite a bit harder: multiple targets possible!

DYNAMIC DISPATCH: GETS COMPLICATED!

A HISTORY OF COMPUTING

Direct Calls

Not so bad

Indirect Calls

Quite a bit harder: multiple targets possible!

```
class A {
  public A f(){ return new C(); }
  public String g(){ return "A"; }
}
class B extends A{
  public String g(){ return "B"; }
}
class C extends A {
  public String g(){ return "C"; }
}
class D extends A {
  public String g(){ return "D"; }
}
class Driver {
  public void main(String[] args){
    A[] aArr = { new A(), new B()};
    for (A a : aArr){
      A res = a.f();
      print(a.g());
      print(res.g());
    }
  }
}
```

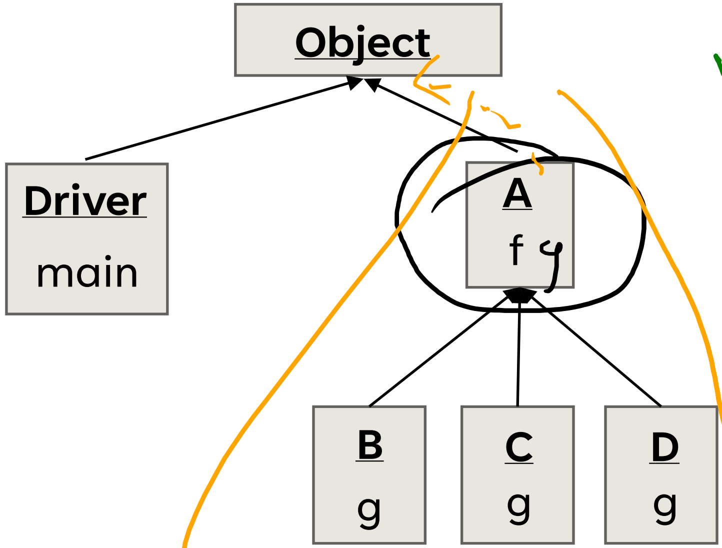
CLASS HIERARCHY ANALYSIS (CHA)

A HISTORY OF COMPUTING

CONSIDER THE SAFE OVER-APPROXIMATION

Treat call as declared type, or any subtype

if(false){
a = new D();
}



On a g call to something with Declared type A:
A.g
B.g
C.g
D.g

```
class A {
  public A f(){ return new C(); }
  public String g(){ return "A"; }
}
class B extends A{
  public String g(){ return "B"; }
}
class C extends A {
  public String g(){ return "C"; }
}
class D extends A {
  public String g(){ return "D"; }
}
class Driver {
  public void main(String[] args){
    A[] aArr = { new A(), new B()};
    for (A a : aArr){
      A res = a.f();
      print(a.g());
      print(res.g());
    }
  }
}
```

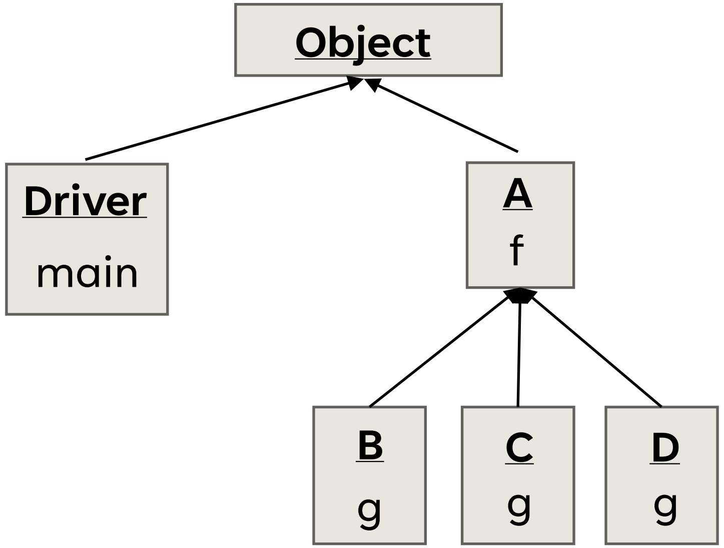
RAPID TYPE ANALYSIS (RTA)

A HISTORY OF COMPUTING

Refinement over CHA

Consider only reachable code

Consider only initialized classes




```java
class A {
  public A f(){ return new C(); }
  public String g(){ return "A"; }
}
class B extends A{
  public String g(){ return "B"; }
}
class C extends A {
  public String g(){ return "C"; }
}
class D extends A {
  public String g(){ return "D"; }
}
class Driver {
  public void main(String[] args){
    A[] aArr = { new A(), new B()};
    for (A a : aArr){
      A res = a.f();
      print(a.g());
      print(res.g());
    }
  }
}
```

RAPID TYPE ANALYSIS (RTA)

A HISTORY OF COMPUTING

Basic Idea: Refinement over CHA

Consider only reachable code

Consider only initialized classes

Consider only reachable code

```
class A {
  public A f(){ return new C(); }
  public String g(){ return "A"; }
}
class B extends A{
  public String g(){ return "B"; }
}
class C extends A {
  public String g(){ return "C"; }
}
class D extends A {
  public String g(){ return "D"; }
}
class Driver {
  public void main(String[] args){
    A[] aArr = { new A(), new B()};
    for (A a : aArr){
      A res = a.f();
      print(a.g());
      print(res.g());
    }
  }
}
```

RAPID TYPE ANALYSIS (RTA)

A HISTORY OF COMPUTING

```
RTA = call graph of functions
CHA = call graph via class hierarchy analysis
W = worklist
W.push(main)
whie not W.empty:
  M = pop W
  T = allocated types in M
  T = T U allocated types in RTA callers of M
  foreach callsite(C) in M
    if C is statically-dispatched:
      add edge C to C's static target
    else:
      M' = methods called from M in CHA
      M' = M' intersect functions declared in T or T-supertypes
      add edge from M to each M'
      W.pushAll(M')
```

RAPID TYPE ANALYSIS (RTA)

A HISTORY OF COMPUTING

AN UNSOUND ANALYSIS!

```
public static Object v;

public static void main(String[] args){
  foo();
  bar();
}

public static void foo(){
  Object o = new A();
  v = o;
}

public static void bar(){
  v.toString();
}
```

RTA will not include an edge from bar to toString because neither bar or its ~~parents~~ callers (main) allocated any instance that toString could be called on

RAPID TYPE ANALYSIS (RTA)

A HISTORY OF COMPUTING

AN UNSOUND ANALYSIS!

```
public static Object v;

public static void main(String[] args){
  Object o = foo();
  bar(o);
}

public static Object foo(){
  return new A();
  v = o;
}

public static void bar(Object o){
  o.toString();
}
```

Call edge to A's toString missing!
Neither bar or its callers (main) allocated a type of A

BEYOND RTA

A HISTORY OF COMPUTING

Assumptions to strengthen analysis

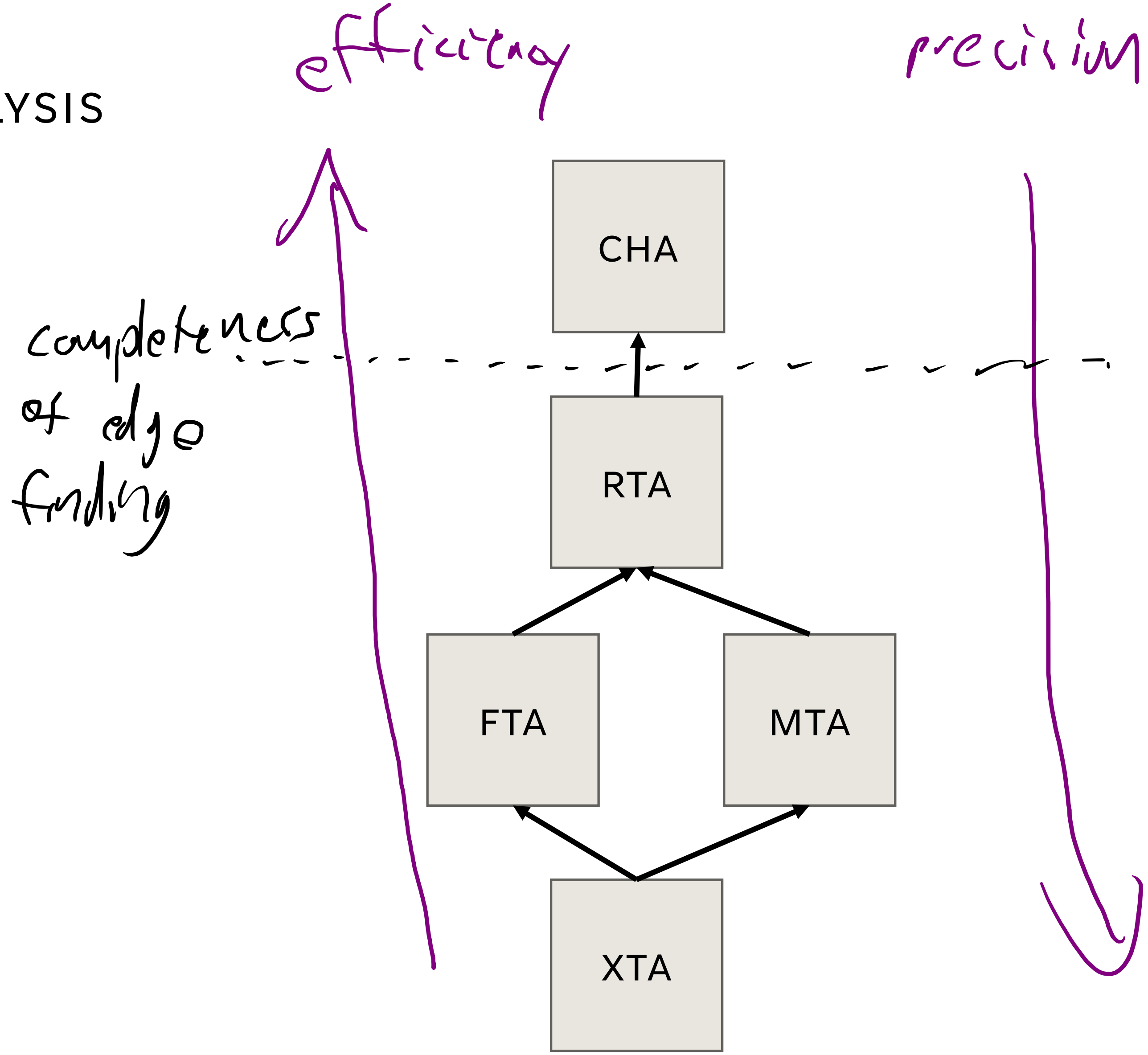
Type safety?

Might not be a safe assumption

FTA adds the constraint that any method that reads from a field can inherit the field-compatible allocated types of any method that could write to that field.

MTA adds the constraint that types allocated in a method and then passed to a method through a parameter should be compatible with the called method's parameter types. MTA also adds the constraint that the return type of each called method be added to the set of allocated types.

XTA: add both the constraints of MTA an FTA



WRAP-UP