ИСП РАН

Разработка и реализация метода поиска дефектов связанных с динамической памятью в бинарном коде

Студент:
Петтик Н.Ю.

Научный руководитель:
Гайсарян С.С.

Обзор дефекта

```
char *ptr = malloc(SIZE);

…

if (error){
    free(ptr);
}

…

printf("%s", ptr);
```

⟶ free(ptr);

Ошибка имеет место при разыменовывании или повторном освобождении указателя на динамическую память, которая была ранее освобождена

Постановка задачи

Разработать и реализовать метод поиска дефектов в бинарном коде:

- Построить модель динамической памяти
- Улучшить алгоритм символьного выполнения в среде BinSide для обработки указателей
- Произвести сравнительную оценку методов межпроцедурного анализа
- Произвести тестирование как на синтетических тестах, так и на реальных проектах

Аналоги

1. GUEB

- Использует дизассемблер IDA
- MonoREIL для анализа переменных
- Межпроцедурный анализ посредством инлайнинга
- 1 проход по циклам
- Написан на Jython

2. LoongChecker

- Улучшенное промежуточное представление eREIL
- VSA алгоритм для нахождения алиасов
- Комбинация с фаззером для подтверждения ошибок
- “Полу-симуляция” для построения аннотаций

Программные средства

IDA PRO:

- Дизассемблирование
- Построение графов потока управления и графа вызовов
- Импортирование в базу данных BinSide

BinSide:

- Трансляция в инструкции RISC архитектуры REIL
- Анализ с помощью фреймворка MonoREIL

Особенности REIL кода

- Не зависит от платформы
- Малый набор инструкций
- Каждая инструкция выполняет ровно одно действие
- Бесконечный набор временных регистров
- Однозначное отображение инструкций

Используемые предположения

- Каждый раз выделяется новый блок динамической памяти
- Соглашение о вызовах fastcall и архитектура x64
- Один проход по циклам
- Предикат ветвления никак не обрабатывается
- Для анализа аннотаций используются эвристики

ИСП РАН

Формальная модель памяти

HE := (id, size) — набор всевозможных значений элементов динамической памяти
HA — набор элементов динамической памяти, которые не были освобождены
HF — набор элементов динамической памяти, которые были освобождены
HeapAccess — каждому элементу из HE ставит в соответствие набор переменных, содержащих указатель на данный элемент

Для функций выделения и освобождения памяти заданы передаточные функции:

$$f_{alloc}(p, HA, HF, HeapAccess, size, id) = (HA', HF, retVal, id')$$

$$retVal = (id, size) \quad HA' = HA(p) \cup \{retVal\} \quad id' = id + 1$$

$$f_{free}(p, HA, HF, HeapAccess, arg) = (HA', HF')$$

$$HA' = HA(p) \backslash HeapAccess(arg) \quad HF' = HF(p) \cup HeapAccess(arg)$$

Так же передаточные функции заданы для инструкций LDM, STM и STR, если их операндами являются элементы из HeapAccess

Символьное выполнение

Вводятся абстрактные символьные элементы:

— Бинарные операции: Addition, Subtraction, Multiplication, BitwiseOr и т.д.

— Унарные операции: Dereference

— Переменные: Symbol, MemoryCell, Literal

— Either - для разрешения объединения значений после ветвления и циклов

Анализ проходит по графу потока управления функции и вызывает передаточные функции для соответствующих инструкций, создавая формулы для выходных значений

Символьное выполнение

Для каждой инструкции хранятся регистры и ячейки памяти и набор формул, являющиеся их значениями

Некоторые формулы упрощаются

Пример

```
add rdi, 4, t0          [rdi = rdi, t0 = rdi+4]
add t0, ssbase, t1      [rdi = rdi, t1 = (rdi+4)+ssbase]
stm rax, , t1           [rdi = rdi, @(rdi+4+ssbase) = rax]
...
ldm t2, , t3            [rdi = rdi, @(rdi+4+ssbase) = rax, t3 = rax]
str t3, , rcx           [rdi = rdi, @(rdi+4+ssbase) = rax, rcx = rax]
```

Внутрипроцедурный анализ

- Поиск адресов функций выделяющих и освобождающих динамическую память
- После вызова функции выделяющей память, в регистр rax помещается символьный элемент Pointer, содержащий размер блока, уникальный номер и статус “не освобожден”
- При вызове функции освобождающей память, указателю в аргументом регистре присваивается статус “освобожден”, или “дважды освобожден”
- Для каждой инструкции загрузки по адресу проверяется статус указателя-операнда
- Указатели со статусами “освобожден” и “не освобожден” объединяются в указатель со статусом “возможно освобожден”

Схема межпроцедурного анализа

1. Граф вызовов разбивается на сильносвязные компоненты
2. В каждой компоненте удаляются обратные ребра
3. Производится топологическая сортировка функций
4. Обход графа вызовов, начиная с листовых функций и заканчивая функцией main
5. При обходе строятся аннотации и происходит внутрипроцедурный анализ

ИСП РАН

Аннотации

- Аннотация на регистры исходной архитектуры
- Аннотация на изменение глобальной памяти
- Аннотация на выделение динамической памяти
- Аннотация на освобождение динамической памяти
- Аннотация на разыменовывания аргумента

Аннотации на функции из стандартной библиотеки задаются вручную перед началом анализа

При вызове функции применяются аннотации к текущему состоянию символьного выполнения

ИСП РАН

Результаты

Набор тестов Juliet

Всего тестов	Содержащих дефект	Истинных срабатываний	Ложных срабатываний
960	480	190	0

ИСП РАН

Результаты

	Всего дефектов	Истинных срабатываний	Ложных срабатываний
gnome-nettool	1	1	1
openjpeg	1	1	3
jasper	1	1	9
accel-ppp	2	2	2

Дальнейшая работа

- Учитывать предикаты ветвления
- Ввести поддержку для остальных архитектур

Заключение

В результате работы было выполнено:

- Разработан и реализован метод поиска уязвимостей в бинарном коде
- Построена модель динамической памяти
- Улучшен алгоритм символьного выполнения в среде BinSide для обработки указателей
- Была выбрана и реализована схема межпроцедурного анализа
- Инструмент был опробован как на синтетических тестах, так и на реальных проектах

Особенности REIL кода

Для работы с памятью используется всего три инструкции:

- **STM reg, , addr** —сохранение регистра по адресу
- **LDM addr, , reg** —загрузка в регистр содержимого по адресу
- **STR reg1, , reg2**—перемещение регистров

Вызов функции осуществляется инструкцией **jcc 1, , func**

Разыменовывание переменной: **LDM @(ptr+ssbase), , reg**

До вызова функции аргумент кладется в регистр rdi/rsi

После вызова, возвращаемое значение из регистра rax сохраняется в память

Недостатки внутрипроцедурного анализа

```
void f(void *p) {
  free(p);
}
```

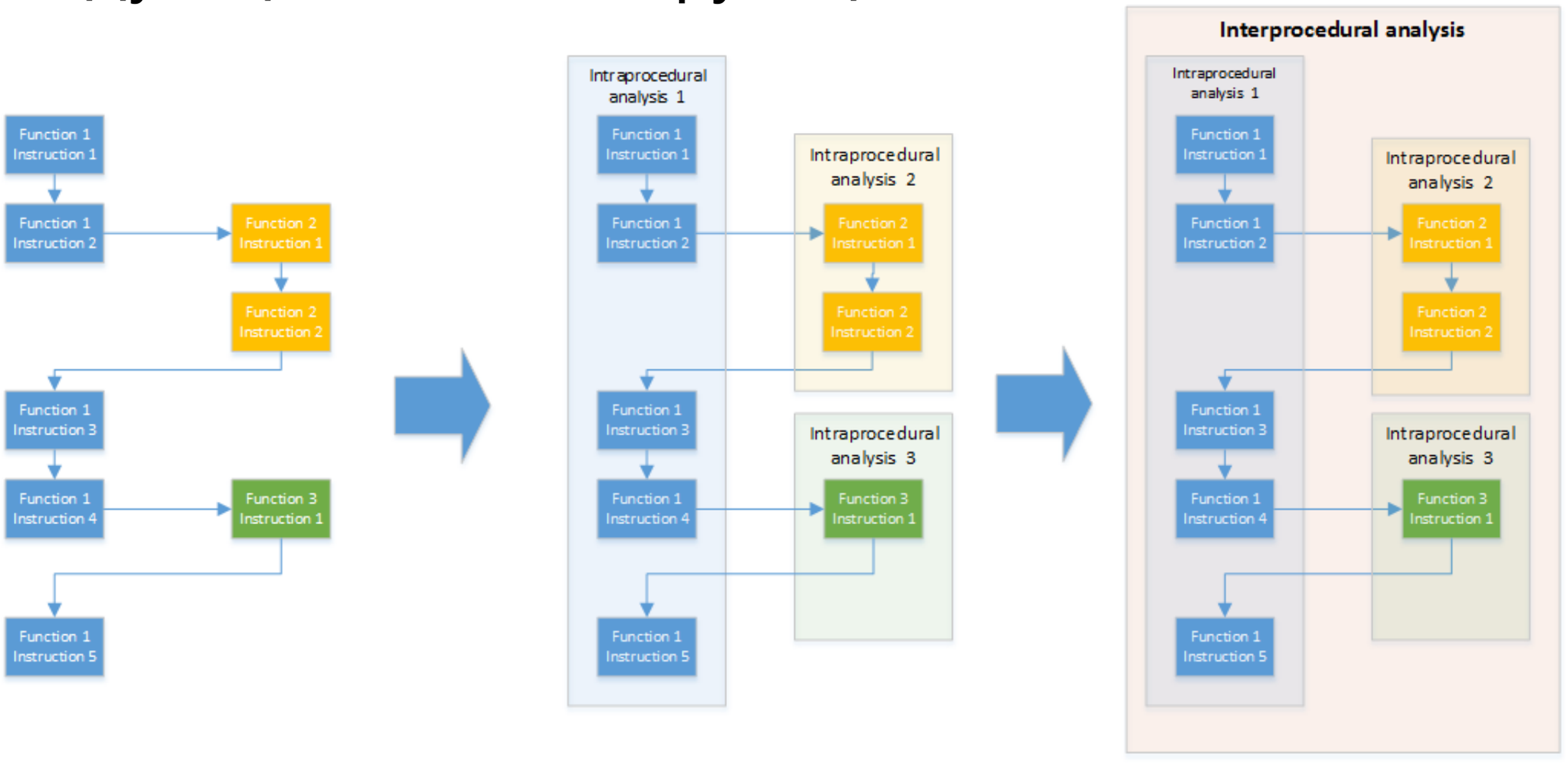
```
void f(void* p) {
  *(int*)p = 7;
 }
```

Невозможно отследить побочные эффекты вызова других функций

Межпроцедурный анализ

Метод аннотаций:

- Каждая функция анализируется один раз
- Создается набор аннотаций, который используется при последующих вызовах функций

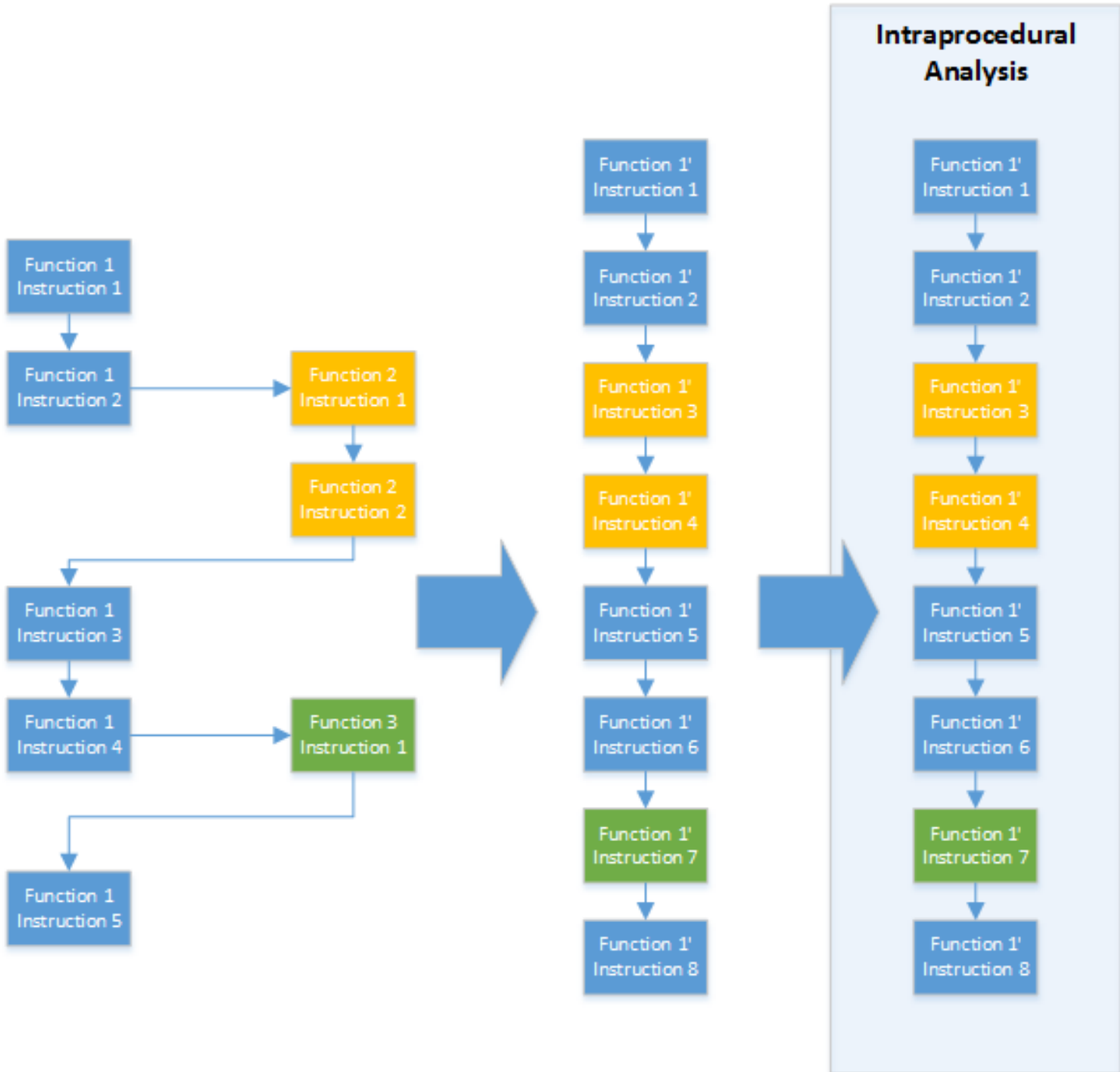


Межпроцедурный анализ

Инлайнинг:

+ Прост в реализации
+ Сохраняет точность анализа

- При каждом вызове функция анализируется заново
- Экспоненциально растут накладные расходы



Анализ аннотаций

Регистры исходной архитектуры:

1. Находим точку возврата функции
2. Если состояние для такой точки содержит регистр исходной архитектуры, то добавляем его в аннотацию
3. Объединяем состояния для всех точек возврата
4. После вызова функции, нужно подставить значения аргументов

Выделение памяти:

1. Находим точку возврата функции
2. Состояние для такой точки содержит регистр, используемый для возвращаемого значение
3. В таком регистре хранится элемент Pointer со статусом “не освобожден”
4. После вызова функции создаем новый элемент Pointer и кладем его в регистр rax