Анализ кода и информационная безопасность

Лекция 13

Анализ обращений к памяти в бинарном коде

МГУ / ВМК / СП

Анализ обращений к памяти

- Обращение к памяти в бинарном коде, как правило, соответствует обращению к некоторому l-значению в исходном коде.
- Примеры l-значений:
  - локальная переменная на стеке;
  - глобальная или статическая переменная в секции данных;
  - переменная в динамически выделенной памяти;
  - часть переменной — поле структуры, элемент массива.
- Проблема: к какому l-значению осуществляет доступ?
- Обращение по конкретному адресу или по смещению относительно регистра стека или фрейма часто может быть легко проанализировано.
- Общий случай сложен.
- Balakrishnan G., Reps T. Analyzing memory accesses in x86 executables.

Условия проведения анализа

1. Рассматривается 32-разрядная процессорная архитектура x86.
2. Проведено дизассемблирование программы:
   - установлены границы подпрограмм;
   - проанализированы обращения к памяти по константным адресам;
   - проанализированы обращения к памяти в стеке.
3. Восстановлен частичный граф вызовов программы:
   - могут быть не восстановлены вызовы по вычисляемом адресам.
4. Восстановлены частичные графа потока управления подпрограмм:
   - могут быть не восстановлены рёбра, соответствующие передаче управления по вычисляемым адресам.
5. Методом сигнатурного поиска распознаны вызовы malloc.

Абстрактный домен

Регионы памяти

- Метод абстрактной интерпретации подходит для того, чтобы проанализировать обращения к памяти по вычисляемым адресам.
- Например, можно воспользоваться интервальной абстракцией, чтобы вычислить множество возможных значений переменных, а, следовательно, и зависящих от них выражений, в том числе адресных.
- Существуют некоторые отличия адресов от «просто значений», которые необходимо учитывать:
  - на одних и тех же адресах в разные моменты времени работы программы могут находиться разные l-значения;
  - одному l-значению могут соответствовать несколько адресов во время работы программы;
  - нельзя статически определить адреса в динамической памяти.

Абстрактный домен

Регионы памяти

Регион памяти — часть адресного пространства программы. Регионы памяти не пересекаются.

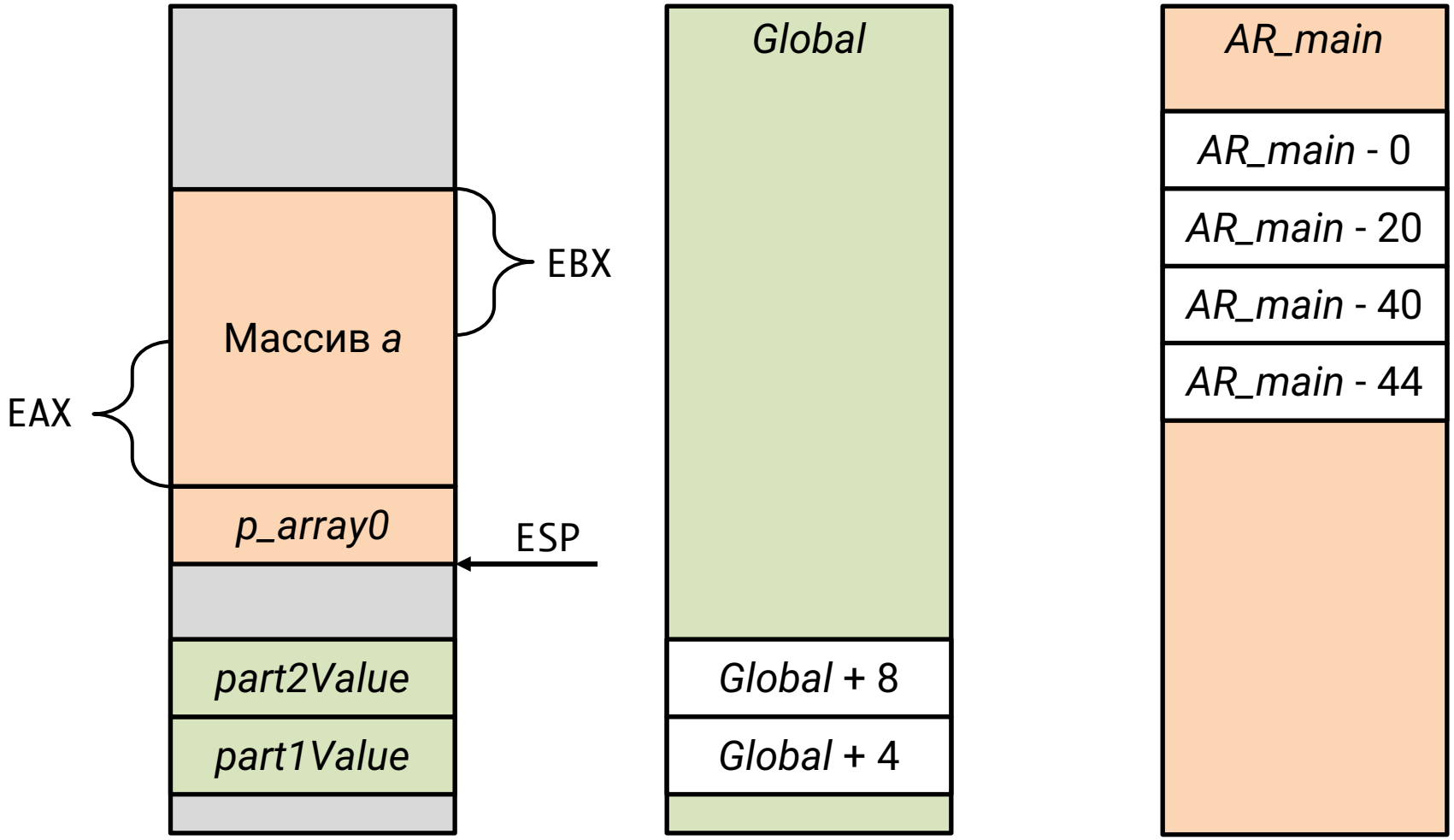
Состав регионов памяти программы:

- один глобальный регион — секции данных программы;
- по одному региону для каждой подпрограммы — фреймы этих подпрограмм;
- по одному региону для каждого вызова функции malloc — участки динамической памяти.

Конкретные адреса регионов памяти и их взаимное расположение не важны. В дальнейшем анализ будет учитывать только смещения в регионах, считая их изолированными.

Абстрактный домен

Регионы памяти



Абстрактный домен

A-loc (абстрактные локации)

A-loc — абстракция над l-значениями, т.е. местами расположения переменных.

A-loc создаётся между известными адресами доступа к памяти:

- в глобальном регионе — между константными адресами;
- в регионе подпрограммы — между различными смещениями во фрейме;
- в malloc-регионе — в начале региона (в общем случае размер такой локации считается бесконечным).

Абстрактный домен

Абстрактные хранилища (abstract stores)

Абстрактное хранилище описывает надмножество возможных адресов, которые могут являться значениями всех a-loc, т.е. абстрактное состояние программы.

При конкретной интерпретации мы могли бы хранить значения вида (*регион*, *смещение*). В абстрактной интерпретации можно использовать какую-либо надаппроксимацию для смещений.

RIC: $(a, b, c, d) : a \times [b, c] + d \equiv \{aZ + d \mid Z \in [b, c]\}$

Пример: конкретное множество {1, 3, 5, 9} описывается RIC-четвёркой (2, 0, 4, 1), которая конкретизируется как {1, 3, 5, 7, 9}.

RIC для всех регионов заданной a-loc будем называть множеством значений данной a-loc (value set).

Абстрактный домен

Множества значений

Множества значений образуют решётку, на которой определены операции:

- $(vs_1 \sqsubseteq vs_2)$ — подмножество;
- $(vs_1 \sqcap vs_2)$ — пересечение (meet) двух множеств;
- $(vs_1 \sqcup vs_2)$ — объединение (join) двух множеств;
- $(vs_1 \nabla vs_2)$ — расширение (widening) первого множества относительно второго;
- $(vs_1 \boxplus c)$ — сдвиг элементов множества на константу;
- $* (vs, s)$ — возвращает два множества a-loc применительно к заданному множеством значений набору адресов и размеру *s* — множество *F* «полностью адресуемых» a-loc и множество *P* «частично адресуемых» a-loc.

Зачем может понадобиться операция расширения?

Алгоритм VSA

Внутрипроцедурный анализ

- Программа приводится к упрощённому виду, где есть следующие типы операторов:
  - *R1 = R2 + c*;
  - *(*R1 + c1*) = *R2 + c2*;
  - *R1 = *(*R2 + c1*) + *c2*;
  - *R1 <= c*;
  - *R1 >= R2*.
- Сначала рассматривается случай, когда в анализируемой функции нет косвенной передачи управления.
- Тогда может быть построен граф потока управления, где вершины соответствуют x86-командам, а рёбра, соответствующие условным переходом, аннотированы соответствующими условиями.
- На этом графе может быть проведена абстрактная интерпретация, для чего необходимо лишь задать передаточные функции.

Алгоритм VSA

Передаточные функции

R1 = R2 + c

let (R2 → vs) ∈ e.Before

e.After := e.Before − [R1 → ∗] ∪ [R1 → vs ⊞ c]

R1 <= c

let [R1 → vsR1] ∈ e.Before and vsc = ([−∞, c], ⊤, . . . , ⊤)

e.After := e.Before − [R1 → ∗] ∪ [R1 → vsR1 ⊓ vsc]

R1 >= R2

let [R1 → vsR1], [R2 → vsR2] ∈ e.Before

and vslb = RemoveUpperBounds(vsR2)

e.After := e.Before − [R1 → ∗] ∪ [R1 → vsR1 ⊓ vslb]

Алгоритм VSA

Передаточные функции

$\underline{*(R1 + c1) = R2 + c2}$

let $[R1 \to vsR1], [R2 \to vsR2] \in e.Before$,

$(F, P) = *(vsR1 \boxplus c1, s)$,

$tmp = e.Before - \{[a \to *] \mid a \in P \cup F\} \cup \{[p \to \top] \mid p \in P\}$,

and Proc be the procedure containing the statement

if ($|F| = 1$ and $|P| = 0$ and (Proc is not recursive) and (F has no heap objects))

then // Strong update

$e.After := (tmp \cup \{[v \to vsR2 \boxplus c2] \mid v \in F\})$

else // Weak update

$e.After := (tmp \cup \{[v \to (vsR2 \boxplus c2) \sqcup vsv] \mid v \in F, [v \to vsv] \in e.Before\})$

Алгоритм VSA

Передаточные функции

$\underline{R1 = *(R2 + c1) + c2}$

let $(R2 \to vsR2) \in e.Before$ and $(F, P) = *(vsR2 \boxplus c1, s)$

if $|P| = 0$

then

let $vsrhs = F \{vsv | v \in F, [v \to vsv] \in e.Before\}$

$e.After := e.Before - [R1 \to *] \cup [R1 \to (vsrhs \boxplus c2)]$

else

$e.After := e.Before - [R1 \to *] \cup [R1 \to \top]$

Алгоритм VSA
Межпроцедурный анализ

- Теперь рассматривается случай, когда в программе несколько процедур, но всё ещё отсутствуют косвенные передачи управления.
- Уточнение a-locs на стеке (PUSH, POP):
  - выявление максимально возможного размера фрейма данной процедуры;
  - создание новых a-locs во всех незадействованных слотах фрейма.
- Формальные параметры на стеке:
  - соответствуют стековым a-loc с положительными смещениями.
- Вызовы и возвраты:
  - обрабатываются на «суперграфе» (~iCFG);
  - требуются передаточные функция для вызова и возврата.

Алгоритм VSA

Передаточные функция вызова и возврата

Передаточная функция вызова

1. Вычисляется объединение (join) абстрактных состояний во всех точках вызова (call sites) вызываемой процедуры *P*.
2. Устанавливается новое абстрактное значение для регистра стека: $(\bot, \ldots, 0, \ldots, \bot)$, где 0 находится на позиции региона процедуры *P*.
3. Производится инициализация значений формальных параметров: копирование значений из региона вызывающей процедуры в регион вызываемой.

Передаточная функция возврата

1. Восстановление значения регистра стека.
2. Обработка alloca и других подобных процедур.

Алгоритм VSA
Аффинные соответствия

- Для того, чтобы итеративный алгоритм сошёлся, в циклах приходится применять операцию расширения.
- В случае, когда обрабатывается какой-либо массив данных, соответствующие указатели получают бесконечную верхнюю границу, что приводит к тому, что применение операции $*$ возвращает лишние a-locs.
- Алгоритм ARA позволяет выявить аффинные соответствия между значениями регистров общего назначения вида $a_0 + \sum_{i=1}^{n} a_i r_i = 0$.
- Информация может быть использована для уточнения:
  - обработки условно выполняемых команд;
  - выполнения расширения в циклах.
- Подробнее: Müller-Olm M., Seidl H. Precise interprocedural analysis through linear algebra.

Алгоритм VSA
Косвенные переходы

- Если программа содержит косвенные вызовы и переходы, то её «суперграф» будет неполным.
- Рассмотрим косвенный переход `J1: JMP [EAX * 4 + 1000]`.
- Пусть абстрактное состояние для EAX содержит запись $\{ [0; 9], \bot, \ldots \}$.
- Возможными адресами перехода будут значения, записанные в ячейках памяти $\{ 1000, 1004, \ldots, 1036 \}$. Далее возможны два случая:
  - если эти адреса относятся к секции кода или константных данных программы, то они могут быть считаны напрямую;
  - если нет, то можно использовать связанные с ними множества значений и добавить все возможные рёбра;
  - отдельно следует рассмотреть случай, когда множество значений в предыдущем варианте — $\top$; тогда разумнее не проводить новые рёбра вовсе.

Алгоритм VSA
Косвенные вызовы

- В целом обрабатываются аналогично переходам.
- Возможно, что целевой адрес относится к середине уже известной функции.
- Возможно, что целевой адрес не относится к уже известной функции. Тогда необходимо возобновить процесс итеративного дизассемблирования с учётом новой информации.
- Следует ли в этом случае перезапустить и алгоритм VSA?

Применения алгоритма VSA

- Значительное улучшение покрытия кода программы при дизассемблировании методом рекурсивного спуска.
- Более точное знание адресов позволяет получать меньшие статические слайсы.
- Алгоритм VSA может использоваться как часть тракта декомпиляции. Применение VSA позволяет получить некоторые сведения о переменных программы и их типах.
- «Побочным эффектом» применения VSA может быть выявление ошибок типа «переполнение буфера».

Применения алгоритма VSA

Алгоритм ASI

- Ramalingam G., Field J., Tip F. Aggregate structure identification and its application to program analysis.
- Задача: выявление на основе шаблонов доступа к памяти в программе структуры используемых данных.
- Постановка задачи в оригинальной статье — для исходного кода со слабой типизацией (COBOL).
- Алгоритм может быть применён к x86-коду (ассемблерный код тоже, несомненно, слабо типизирован), но предварительно необходимо получить информацию об указателях (points-to), а также диапазоны (range) и шаги (stride) для всех переменных.
- Такая информация может быть получена в результате применения сначала алгоритма VSA.

Литература к лекции

Основные источники

1. Balakrishnan G., Reps T. Analyzing memory accesses in x86 executables // International conference on compiler construction. – Springer Berlin Heidelberg, 2004. – С. 5-23.
2. Müller-Olm M., Seidl H. Precise interprocedural analysis through linear algebra // ACM SIGPLAN Notices. – ACM, 2004. – Т. 39. – №. 1. – С. 330-341.

Литература к лекции

Дополнительные источники

1. Ramalingam G., Field J., Tip F. Aggregate structure identification and its application to program analysis // Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – ACM, 1999. – С. 119-132.