А.В. Макаров, С.Ю. Скоробогатов, А.М. Чеповский

Учебный курс “CIL и системное программирование в Microsoft .NET”

Лекция 15.

Верификация CIL-кода

Введение

- Говорят, что код программ не разрушает память, если эти программы, будучи запущенными в одном адресном пространстве, корректно изолированы друг от друга
- Верификация кода – это процесс автоматического доказательства того, что этот код не разрушает память

15.1. Классификация применяемых на практике алгоритмов верификации

- Алгоритмы верификации являются метавычислительными алгоритмами. Поэтому совершенно неудивительно, что алгоритмы верификации в общем случае имеют проблемы, свойственные любым метавычислительным алгоритмам, а именно:
  - экспоненциальная сложность
  - потенциальная нетерминируемость
- Такие алгоритмы представляют скорее теоретический, нежели практический интерес. Поэтому их разработчикам приходится каким-то образом ограничивать задачу, чтобы получить применимые на практике результаты

Два подхода к ограничению задачи для алгоритмов верификации

- Концентрация усилий на выявлении только таких ошибок в программах, поиск которых не требует экспоненциальной сложности и выполняется за конечное время
  - Для этого подхода характерно то, что существуют программы, в которых верификатор не находит ни одной ошибки, и которые, тем не менее, разрушают память
- Работа лишь с некоторым подмножеством программ, для которых факт неразрушения ими памяти можно доказать за конечное время с использованием алгоритма приемлемой сложности
  - Этот тип верификаторов отличается тем, что существуют не разрушающие память программы, которые не могут быть доказаны алгоритмом

Вопрос применимости алгоритма верификации с ограничениями

- Если задача некоторого метавычислительного алгоритма была как-то ограничена, то сразу же возникает вопрос о том, какие входные данные (программы) он сможет обработать
- Для суперкомпиляторов и частичных вычислителей этот вопрос до сих пор остается открытым (ответ на него приходится искать методом проб и ошибок)
- Для верификаторов ответ зависит от того, какой подход (первый или второй) был выбран для ограничения задачи

Вопрос применимости алгоритма верификации с ограничениями (первый тип)

- Верификаторы, предназначенные для выявления определенного класса ошибок в программах, используются при разработке программ на этапе тестирования. Тем самым, на вход таких верификаторов подаются любые программы в надежде, что в них будут обнаружены ошибки
- Запуск такого верификатора можно рассматривать как один из тестов, которым подвергается программа. Если в некоторой программе верификатор не обнаружил ни одной ошибки, то можно считать, что его запуск был напрасным

Вопрос применимости алгоритма верификации с ограничениями (второй тип)

- Верификаторы, способные доказать, что программа, принадлежащая некоторому классу программ, не разрушает память, служат для обеспечения безопасности
- Использование таких верификаторов оправдано и удобно в тех случаях, если существуют специальные компиляторы, генерирующие такой код, который не только заведомо не разрушает память, но и гарантированно проходит верификацию
- Тем самым ответ на вопрос о том, для каких программ верификатор может доказать, что они не разрушают память, становится тривиально прост: для тех программ, которые были откомпилированы специальным компилятором

15.2. Особенности верификатора кода, используемого в .NET

- Верификатор кода .NET относится ко второму типу. Он может доказать то, что сборки, генерируемые компиляторами C#, J# и Visual Basic .NET, не разрушают память. При этом он терминируем и имеет линейную сложность

Зачем в .NET нужен верификатор

- Здесь может возникнуть следующий вопрос: зачем нужен верификатор, если компиляторы и так генерируют не разрушающий память код? Чтобы ответить на этот вопрос, нужно вспомнить, что:
  - существуют компиляторы (например, Visual C++ with Managed Extensions), которые в общем случае могут порождать код, разрушающий память
  - в языке C# предусмотрены unsafe-блоки и unsafe-методы, внутри которых может находиться код, способный разрушить память
  - любой CIL-код, в том числе и разрушающий память, можно непосредственно компилировать с помощью ILASM
  - всеми этими возможностями может воспользоваться злоумышленник для написания вредоносного кода

Верифицированный код и верифицируемый код

- Верифицируемый код – это код, для которого верификатор может доказать, что он не разрушает память. Другими словами, для верифицируемого кода можно заранее, еще до запуска верификатора, сказать, что он успешно пройдет верификацию (такой код может генерироваться компилятором C#)
- Верифицированный код – это код, для которого верификатор доказал, что он не разрушает память. То есть чтобы из верифицируемого кода получить верифицированный код, нужно обязательно запустить верификатор

15.3. Алгоритм верификации

- В спецификации CLI предложен базовый вариант алгоритма верификации. Любая реализация CLI должна включать верификатор, верифицирующий по крайней мере те программы, которые допускает базовый алгоритм
- Алгоритм верификации представляет собой вариант абстрактного интерпретатора CIL-кода. Линейность алгоритма верификации достигается за счет того, что он последовательно просматривает тело метода инструкция за инструкцией (при этом ни одна инструкция не обрабатывается дважды)

15.3.1. Совместимость типов

- Пусть S и T – типы. Тогда S[] и T[] – соответствующие им массивные типы, а S& и T& – типы соответствующих управляемых указателей
- Тот факт, что S совместим по присваиванию с T, мы будем записывать как S := T
- Операция := рефлексивна и транзитивна
- Правила совместимости типов:
  - S := T, если S – базовый класс для T или интерфейс, реализуемый T, и при этом T – не является типом-значением
  - S := T, если S и T – интерфейсы, и реализация T требует реализации S
  - S := null, если S – объектный тип или интерфейс
  - S[] := T[], если S := T и размерности массивов совпадают
  - S& := T&, если S := T
- Если ни одно из этих правил не выполняется, то типы S и T – несовместимы

15.3.2. Конфигурации стека

- Будем называть конфигурацией стека данные о количестве слотов на стеке и типы значений, лежащих в этих слотах. При этом конфигурацию, содержащую 0 слотов, будем называть пустой конфигурацией
- Рассмотрим две операции над конфигурациями, которые используются в алгоритме верификации:
  1. проверка совместимости двух конфигураций
  2. слияние двух конфигураций

Проверка совместимости двух конфигураций

- Операция проверки совместимости имеет два операнда: конфигурации C и K
- Она возвращает булевское значение, показывающее, совместима ли конфигурация C с конфигурацией K

Алгоритм вычисления операции совместимости двух конфигураций

- Если количество слотов в конфигурациях C и K различно, то возвращаем false. В противном случае пусть N – количество слотов в конфигурации C (естественно, и в K тоже)
- Если N = 0, то возвращаем true
- Пусть i пробегает значения от 1 до N. Тогда для каждого такого i выполняем следующее:
  - пусть S – тип i-го слота конфигурации C, а T – тип i-го слота конфигурации K
  - Если не T := S, то возвращаем false
- Возвращаем true

Операция слияния двух конфигураций

- Операция слияния двух конфигураций также имеет два операнда: конфигурации C и K
- Она может либо закончиться неуспехом, либо возвращает результирующую конфигурацию R

Алгоритм слияния двух конфигураций

- Если количество слотов в конфигурациях C и K различно, то алгоритм завершается неуспехом. В противном случае пусть N – количество слотов в конфигурации C (естественно, и в K тоже)
- Если N = 0, то возвращаем пустую конфигурацию

(продолжение на следующем слайде)

Алгоритм слияния двух конфигураций (продолжение)

- Пусть i пробегает значения от 1 до N. Тогда для каждого такого i выполняем следующее:
  - пусть S – тип i-го слота конфигурации C, а T – тип i-го слота конфигурации K.
  - вычисляем тип U i-го слота результирующей конфигурации:
    - если S := T, то U = S.
    - в противном случае, если T := S, то U = T.
    - в противном случае, если T и S – объектные типы и у них существует ближайший базовый тип V, то U = V.
    - в противном случае, алгоритм завершается неуспехом.
- Возвращаем результирующую конфигурацию

15.3.3. Описание алгоритма

- В процессе работы алгоритма для каждой инструкции CIL вычисляется конфигурация стека. При этом фактические значения, лежащие на стеке, в локальных переменных и параметрах метода, не учитываются
- Возможны два результата работы алгоритма:
  - Успешная верификация
    - Метод не содержит неверифицируемых инструкций. Все возможные пути передачи управления в теле метода рассмотрены. Для каждой инструкции вычислена конфигурация стека
  - Неуспешная верификация
    - Метод либо содержит неверифицируемые инструкции, либо в процессе вычисления конфигурации стека для некоторой инструкции было выявлено противоречие

Данные, с которыми работает алгоритм

- Неизменяемые данные:
  - Метаданные сборки, в том числе количество и типы локальных переменных и параметров метода
  - Массив P, содержащий инструкции CIL в том порядке, в каком они записаны в теле метода. Пусть N – количество инструкций
- Изменяемые данные:
  - Массив M размера N, в котором хранятся вычисляемые в процессе верификации конфигурации стека

Верхний уровень алгоритма

- В начале работы алгоритма в массиве M не записано ни одной конфигурации.
- Алгоритм последовательно просматривает массив P, то есть на каждом шаге работает с одной инструкцией P [i], где i пробегает значения от 1 до N (будем считать, что массивы P и M нумеруются, начиная с единицы)

Шаг алгоритма

- Итак, P[i] – текущая рассматриваемая инструкция. Тогда смотрим, что собой представляет M[i]. Если M[i] еще не содержит конфигурацию стека, то записываем в M[i] пустую конфигурацию
- Проверяем, верифицируема ли инструкция P[i], учитывая конфигурацию стека M[i] и информацию, содержащуюся в метаданных. Если инструкция в принципе не верифицируема или не может быть выполнена при заданной конфигурации стека, то алгоритм завершается неуспехом
- Осуществляем абстрактное выполнение инструкции P [i]. Другими словами, вычисляем конфигурацию стека S, которая получилась бы после реального выполнения инструкции

(продолжение на следующем слайде)

Шаг алгоритма (продолжение)

- Определяем, на какие инструкции может быть передано выполнение после инструкции P[i]. Для каждой такой инструкции P[j] выполняем следующее:
  - если j < i, то проверяем, совместима ли конфигурация S с конфигурацией M[j]. Если оказывается, что несовместима, то алгоритм завершается неуспехом.
  - если j >= i и M[j] еще не содержит конфигурации стека, то M [j] = S.
  - если j >= i и M[j] уже содержит конфигурацию стека, то выполняем попытку слияния конфигураций S и M[j]. Если попытка увенчалась успехом, то записываем результат слияния в M[j]. В противном случае алгоритм завершается неуспехом