Статический анализ бинарных модулей среды z/OS с помощью Java toolkit for static analysis of binaries (Jakstab)

Миронович В. С., студент кафедры информатики СПбГУ,
basilmiron@gmail.com

Аннотация

В данной работе рассматривается реализация статического анализа бинарных модулей операционной системы мейнфреймов z/OS с помощью Jakstab — платформы для статического анализа исполняемых файлов архитектуры x86. Jakstab был разработан в рамках диссертации на соискание степени PhD немецкого исследователя Йоханнеса Киндера.

Введение

Статический анализ программ — анализ производимый без исполнения самой программы. Позволяет выявлять некоторые недочёты и слабые места программы, например, такие как присутствие мёртвого кода, и улучшать качество кода, например, оптимизировать циклы.

При выполнении статического анализа бинарного кода возникают следующие трудности:

- наличие регистровых переходов
- отсутствие типизации
- арифметика указателей
- сложная семантика команд процессора

Наличие косвенных переходов делает возможным переход в любую точку программы, поэтому для построения информативного графа потока управления (без многочисленных избыточных рёбер) в качестве адреса перехода нужно получать осмысленный набор значений, а для этого необходимо выполнять анализ потока данных. В свою очередь анализ потока данных невозможен без корректно выполненного анализа потока управления, что приводит нас к проблеме «замкнутого круга».

Поэтому для реализации статического анализа бинарных модулей среды z/OS достаточно естественно воспользоваться существующим

анализатором бинарного кода, который решает хотя бы часть из вышеобозначенных проблем, и самостоятельно разрабатывать только модуль, описывающий конкретную архитектуру (z/Architecture).

У вышеназванной архитектуры есть ряд особенностей, которые нужно учитывать при реализации модуля:

- большое количество форматов команд: на данный момент существует 29 базовых форматов, у некоторых из них есть подформаты
- наличие служебного регистра процессора Program Status Word (PSW), который содержит набор управляющих полей, например:
  1. Condition Code (CC): 18-й и 19-й биты PSW, по сути, — один двухбитовый флаг, который устанавливается после выполнения арифметических, логических и некоторых других операций, и на основе которого реализуются все условные переходы (в отличие от CF, OF, SF, ZF — четырёх битовых флагов в x86)
  2. Basic Addressing Mode (BA): 31-й и 32-й биты PSW, отвечает за установку одного из трёх режимов адресации (то есть контролирует длину адреса)
- наличие команд, семантику которых можно задать только зная конкретные операнды

Архитектура Jakstab

Jakstab — фреймворк, строящий, если это возможно, корректный граф потока управления и осуществляющий статический анализ исполняемых файлов. Jakstab поддерживает только архитектуру процессора x86, но позволяет достаточно естественным образом добавить поддержку других архитектур[2].

Jakstab «на лету» транслирует машинный код в низкоуровневый промежуточный язык RTL, так как параллельно выполняется анализ потока данных на растущем графе потока управления[1]. Информация, получаемая при анализе потока данных, используется для вычисления адресов переходов и вычисления следующих значений программного

счётчика. Унифицированная архитектура дизассемблирования и анализа хороша тем, что не зависит от внешнего дизассемблера и позволяет использовать результаты статического анализа для более точного построения графа потока управления.

Итерации цикла дизассемблирования и анализа (см. Рис. 1) состоят из:

- получение объекта инструкции x86 с конкретными операндами путём декодирования её из бинарного кода
- отображение объекта инструкции x86 в соответствующую ей инструкцию Pentium — этот шаг обусловлен тем, что семантики инструкций, представленные в Jakstab, основаны на SSL-определениях[1] для команд процессора Intel Pentium, являющихся частью декомпилятора Boomerang[5], а набор базовых операторов промежуточного языка (IL RTL) во многом аналогичен SSL[4]
- получение семантики, описанной на промежуточном языке, для соответствующей инструкции
- преобразование последовательности операторов промежуточного языка в набор рёбер автомата потока управления[2] (CFA[6])
- абстрактная интерпретация рёбер CFA в набор состояний программы (известные значения регистров, памяти и флагов)
- вычисление, если возможно нового значения программного счётчика
- переход к смещению внутри бинарного модуля, считывание очередной инструкции

Реализация прототипа модуля, описывающего z/Architecture

Для того чтобы статически анализировать с помощью Jakstab бинарные модули среды z/OS, был реализован прототип[3] модуля, описы-

[1] Semantic Specification Language

[2] Представление программы в виде графа, в вершинах которого логические состояния, а все операции помещены на рёбра

[3] Пока что только прототип, так как полное описание z/Architecture — очень трудоёмкая задача вследствие сложности архитектуры мейнфреймов (большое коли-

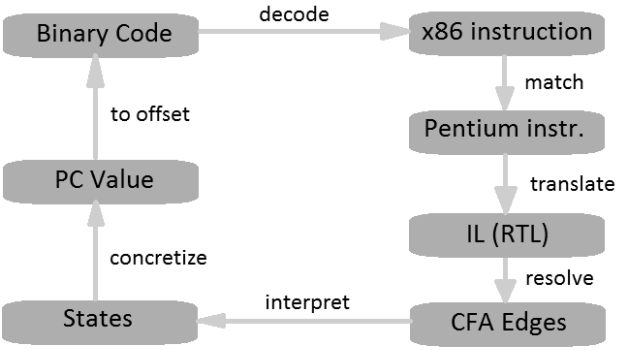

Рис. 1: Интегрированный цикл дизассемблирования и анализа

вающего z/Architecture, то есть набор машинных команд с их семантиками, набор регистров, флаги и режим адресации.

Важным для реализации понятием является формат инструкции[3]. Он задаёт:

- размер инструкции (2, 4 или 6 байт)
- размер кода операции (1 или 2 байта)
- количество и порядок операндов
- тип каждого операнда (регистр, адрес, immediate-значение или маска)

В рамках прототипа были описаны 10 (считая подформаты и 2 фиктивных, созданных для удобства реализации) наиболее популярных форматов.

Модуль, описывающий z/Architecture, состоит из трёх частей:

- дизассемблер, который создаёт объект инструкции с декодированными операндами
- иерархия классов инструкций и их операндов

чество команд с нетривиальной семантикой, разные режимы адресации, supervisor call-ы и т.д.)

- транслятор, который объекту инструкции сопоставляет последовательность операторов промежуточного языка

Встраивание разработанного модуля делается естественным образом, путём замены соответствующих частей Jakstab2.

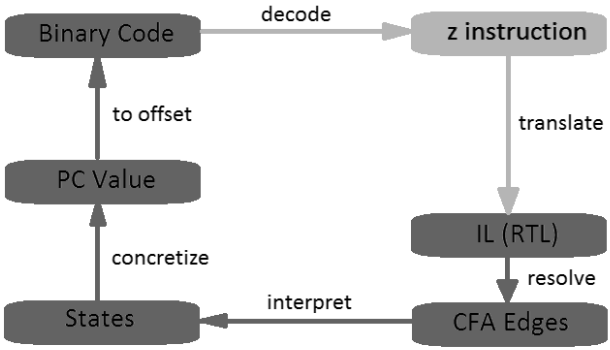

Рис. 2: Цикл дизассемблирования и анализа после встраивания разработанного модуля

На выходе получаем:

- автомат потока управления
- граф потока управления
- ассемблерный листинг

Сценарии использования и перспективы развития

Полученные результаты в дальнейшем можно будет использовать для:

- Понимания программ (program understanding), для которых нет хорошей документации и/или отсутствует исходный код. Подобные ситуации возможны, вследствие наличия обширного legacy (длительной поддержки программного обеспечения для мейнфреймов)

- Реализации различного статического анализа с помощью полученного графа потока управления, например:
    1. проверка на реентерабельность
    2. сбор начальной информации (подозрительных участков кода) для динамического детектора состояний гонки

Заключение

В настоящий момент прототип модуля содержит описание небольшого, концептуально важного подмножества z/Architecture (например, есть возможность корректно обрабатывать условные и косвенные переходы), и остаётся ещё много деталей архитектуры и команд, которые должны быть обработаны, но сделана важная часть работы — встраивание в Jakstab и тестирование на идеологически важных примерах. Далее, в основном, следует дорабатывать прототип по уже известной, описанной схеме.

Литература

[1] Dipl.-Inf. Johannes Kinder: Static Analysis of x86 Executables

[2] The Jakstab static analysis platform for binaries https://github.com/jkinder/jakstab

[3] z/Architecture Principles of Operation, Tenth Edition

[4] Cristina Cifuentes, Shane Sendall: Specifying the Semantics of Machine Instructions https://pdfs.semanticscholar.org/f2a9/173d93016af42da4fa1e9671233bb338a73c.pd

[5] The Boomerang Decompiler Project http://boomerang.sourceforge.net/

[6] Lazy Abstraction: Control flow automata for C programs http://cseweb.ucsd.edu/ rjhala/papers/lazy_abstraction.pdf