РАССМОТРЕНИЕ ПРЕДСТАВЛЕНИЙ КОДА ПРОГРАММЫ С ПОЗИЦИИ МЕТАДАННЫХ

Израилов К.Е.

В условиях повсеместного использования систем с высоким требованием к уровню безопасности, их программному обеспечению уделяется особое внимание. В частности, оно не должно содержать недокументированных возможностей (НДВ), способных нанести вред системе. Для поиска НДВ может применяться процесс декомпиляции исполняемого кода, позволяющий впоследствии производить анализ алгоритмов и поиск уязвимостей вручную.

Принцип процесса декомпиляции сводится к получению кода, аналогичного начальному, на высокоуровневом языке из его машинного представления. Данный процесс можно назвать “аппроксимацией” кода, поскольку он преобразует сложное для анализа человеком представление в более простое. Задача декомпиляции сложна для реализации, так как она не является чисто линейной и оперирует предсказаниями (по поводу начального вида программы) в ходе своего решения. В отличие от нее, прямой процесс – компиляция – является линейным и достаточно хорошо реализуемым. В рассмотрение не берутся языки программирования, код на которых в конечном представлении содержит всю информацию о начальном (например, набор языков .Net).

Как известно, для решения сложных задач используется универсальный прием – декомпозиция, то есть разбиение одной задачи на несколько задач-модулей. Декомпозиция состоит в четком определении функций каждого модуля, а также порядка их взаимодействия. В результате достигается логическое упрощение задачи, а, кроме того, появляется возможность модификации отдельных модулей без изменения остальной части.

Данный прием хорошо применим и для текущей задачи. Процесс декомпиляции разбивается на последовательность алгоритмических модулей.

В качестве таких модулей выступает поэтапное преобразование кода в различные представления, используемые в процессе компиляции.

Код на всем этапе преобразования от исходного до машинного имеет следующие представления:

- высокоуровневое (начальное), используемое разработчиками программного обеспечения;
- промежуточное, используемое в процессе сборки (данное представление зависит от конкретной реализации компилятора);
- низкоуровневое (ассемблер), содержащее мнемонические обозначений машинных инструкций;
- машинное (конечное), содержащее последовательность байт, выполняемых напрямую на платформе.

Таким образом, компиляция есть преобразование кода от начального представления к конечному, а декомпиляция, соответственно – обратный процесс.

Рассмотрим различия между представлениями с точки зрения метаданных, содержащихся в них. Метаданные в общем смысле являются структурированными данными, представляющими собой характеристики описываемых данных. Применительно к коду программ, метаданными являются: способ ее структурированного представления, информация о свойствах объектов и описания алгоритмов.

Исполняемый код можно представить как линейный набор инструкций, переходов между ними и метаданных, отражающих используемый язык программирования. Поскольку язык программирования является способом передачи команд и данных от человека к компьютеру, то метаданные являются той частью представления программы, которая позволяет человеку воспринимать ее в понятных образах и терминах (число, объект, операция, повторение, входные и выходные параметры, группа, подобие и т.п.). В процессе компиляции часть метаданных теряется по причине их

ненадобности для исполнения машинного кода. Отсюда, процесс декомпиляции можно рассматривать как процесс их восстановления.

Высокоуровневое представление содержит максимальное количество метаданных программы, а именно:

- полная информация о типах объектов (целочисленные типы различных размеров, типы с плавающие точкой и т.п.);
- информация о способах доступа к объектам (константные объекты и т.п.);
- наличие функций, то есть процедур с входными и выходными параметрами;
- наличие операций для описания структурированных алгоритмов (циклы, ветвления и т.п.);
- структурированный вид алгоритмов, то есть с ответствующими безусловными переходами;
- другая информация, используемая в основном для удобства программирования (конструкторы/деструкторы, шаблоны, виртуальные функции и т.п.).

Промежуточное представление зависит от конкретной реализации компилятора. Как правило, по сравнению с высокоуровневым оно содержит те же метаданные, но в виде, удобном компилятору (таблицы символов, внутренние графы и деревья).

Низкоуровневое представление представляет собой данные и процедуры в виде набора мнемонических обозначений машинных инструкций. Оно содержит минимальный набор следующих метаданных, необходимых для получения исполняемого кода:

- именованные данные заданного размера (иногда и структуры);
- процедуры;
- интуитивно-именованные элементарные команды процедур;
- другая общая информация, такая как секции данных, способ доступа и т.п.

Машинное представление является конечным. Код на нем представляет собой последовательность байт, выполняемых напрямую на платформе. Метаданные отсутствуют в связи с отсутствием необходимости в дальнейшем преобразовании кода и по причине оптимизации данного представления для выполнения.

Приведем пример сравнения метаданных в различных представлениях (см. табл. 1). В качестве программы выбрана функция “test()”, бесконечно инкрементирующая глобальную переменную “count”.

Таблица 1. Сравнение метаданных в различных представлениях

Представление	**Пример**	**Метаданные**

Код в высокоуровневом представлении	// C/C++ Language
	int count = 0;
	void test(void){
	  while(1){
	  ++count;
	  }
	}
		Сигнатура функции test – функция не имеет входных и выходных параметров.
		Алгоритм функции содержит цикл.
		Цикл в теле функции является бесконечным.
		Результатом работы итерации цикла является инкрементирование переменной count.
		Код программы имеет структурированный вид; все переменные и функции имеют четко-определенные сигнатуры.

Код в промежуточном представ-	Data: count; Function: test;	Те же, что и в высокоуровневом представлении. Ме-

лении	Function in_arg: void; Function out_arg: void; Function.while_condition: 1;	таданные более удобны для обработки компилятором, чем человеком.

Код в низкоуровневом представлении	// Assembler Language
	.global count
	count:
	.size count, 4
	.zero 4
	.section ".text"
	.type test, @function
	test:
	stwu 1,-16(1)
	stw 31,12(1)
	mr 31,1
	.L2:
	lis 0,count@ha
	mr 9,0
	lwz 0,count@l(9)
	addic 9,0,1
	lis 0,count@ha
	mr 11,0
	stw 9,count@l(11)
	b .L2
		Переменная count размера 4 (тип long), инициализированная 0-м значением.
		Функция test.
		Функция содержит алгоритм (описан на мнемонических командах процессора).
		Присутствует метка (.L2) и переход на нее (b .L2).
		Код программы имеет линейный вид с выделенной секцией данных и процедур.

Код в машинном представлении	// Machine code (PowerPC) 9421fff093e1000c 7c3f0b783c008000 7c0903788009002c 312000013c008000	Отсутствуют.

	7c0b0378912b002c 4bffffe400000000	

Код в представлении языка, метаданными в котором являются его составной частью	// C# Language
	class Program
	{
	static int count = 0;
	static void Test()
	{
	while (true)
	{
	++count;
	}
	}
	}
		В данном коде метаданные являются его составной частью. Они содержат всю информацию об объектах и выполняемых кодом действиях.

Выделим следующие группы метаданных и опишем способы их восстановления (см. табл. 2).

Таблица 2. Способы восстановления метаданных

Название	**Описание и способ восстановления**
Числовые типы объектов	Информация о типах объектов может быть получена на основании использовании памяти, выделенной под этот объект.
Сигнатура функций (входные/выходные параметры)	Параметры могут быть определены по первому использованию (входные параметры, значение их присваивается перед вызовом функции) и последнему присваиванию (выходные параметры, значение их используется после вызова функции) регистров в теле функции.
Условные пере-	Условные переходы и соответствующие им блоки могут

ходы	быть выделены на основании инструкций условных переходов и анализа графа потока управления.
Циклы	Циклы могут быть выделены на основании анализа графа потока управления и специфичных для них шаблонов (главный вход, условный выход, инкрементируемая переменная и т.п.)
Вызовы функций	Информация о вызовах функций и передаваемых в нее параметрах может быть получена с помощью ранее восстановленных сигнатур функций и инструкций вызова.
Высокоуровневые структурные элементы	Информация о таких сущностях языка, как классы, виртуальные функции, конструкторы и т.п., может быть выделена в крайне редких случаях по причине сильной неоднозначности в критериях их наличия в коде. В качестве результата работы декомпилятора она не рассматривается.
Внешняя информация	Некоторая информация о программе может быть получена с помощью других утилит (например, дополнительная информация из программы IDA). Она может быть использована для корректировки анализа декомпилятора.

В результате рассмотрения можно утверждать о том, что такое последовательное представление кода позволяет упростить реализацию алгоритма декомпиляции. Также можно прогнозировать хорошее соответствии восстанавливаемого кода исходному за счет использования объектов и этапов, применяемых в прямом процессе – компиляции.