Восстановление потока управления в точках виртуальных вызовов из биткода LLVM

Избышев Алексей Олегович

Аспирант

Институт системного программирования РАН, Москва, Россия

E-mail: izbyshev@ispras.ru

В сферах деятельности, связанных с исследованием низкоуровнего представления программы (таких, как обратная инженерия или анализ бинарного кода с целью проверки его соответствия поставленным требованиям), возникает задача восстановления потока управления. Если языком программирования является C++, то программа, как правило, содержит большое число вызовов виртуальных функций. При компиляции каждый виртуальный вызов превращается в последовательность инструкций, выполняющих получение из объекта указателя на таблицу виртуальных функций (ТВФ), получение адреса вызываемой функции из ТВФ по индексу и вызов по полученному адресу. При восстановлении потока управления для каждого такого непрямого вызова необходимо уметь строить множество функций, которые потенциально могут быть вызваны. В рамках статического анализа такая задача обычно решается на уровне исходного кода[1], однако подход, предложенный в данной работе, работает на уровне биткода LLVM[2], используя тот факт, что биткод является типизированным.

Каждый указатель на функцию, использующийся для виртуального вызова, имеет тип, из которого можно получить тип T первого аргумента. По соглашению о вызовах[3], T является верхней границей динамического типа объекта-адресата вызова. Тогда можно утверждать, что вызываемая функция входит в множество V всех виртуальных функций, определённых в потомках T (включая сам T) и имеющих такой же индекс в частях ТВФ, относящихся к типу T, как и используемый в рассматриваемом вызове. Вычислению множества V для каждой точки виртуального вызова и посвящена данная работа. Заметим, что V является надмножеством множества функций, которые на самом деле могут быть вызваны в данной точке, поэтому предложенный подход разумно использовать, когда более точные методы не дают результата.

Для построения множества V разработан алгоритм, состоящий из нескольких этапов. Во-первых, необходимо восстановить иерар-

хию классов программы. Для этой цели используются структуры данных, содержащие информацию о типах (RTTI), по умолчанию генерируемые компилятором для всех классов, имеющих ТВФ. Во-вторых, необходимо построить соответствие между ТВФ и типами данных LLVM. В работе показано, что в общем случае это невозможно, и предложен алгоритм, позволяющий сделать это для типов, выступающих в качестве типа первого аргумента виртуальных вызовов, благодаря типизации указателей на функции в биткоде. Наконец, для получения функции по индексу необходимо восстановить структуру ТВФ. ТВФ имеет сложную структуру и является набором массивов, содержащих указатели на функции и дополнительную информацию, причём каждый массив соответствует одному или нескольким объектам базовых классов в составе объекта производного класса. Заметим, что построение ТВФ не может быть выполнено по тому же алгоритму, что и в компиляторе, так как в биткоде отсутствует нужная для этого информация. Предложен альтернативный алгоритм, использующий для разрешения неоднозначностей служебную структуру данных (VTT), генерируемую для классов, имеющих виртуальные базовые классы.

Кроме того, рассмотрен вопрос неполноты информации о классах в анализируемой программе, возникающей из-за удаления неиспользуемых символов (в частности, виртуальных таблиц) при оптимизации. Показано, что в общем случае определить факт неполноты невозможно, и предложены модификации упомянутых выше алгоритмов, позволяющие обнаруживать неполноту в случаях, когда это влияет на правильность их работы.

Литература

1. Bacon D. F. and Sweeney P. F. Fast Static Analysis of C++ Virtual Function Calls // In Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, San Jose, USA, 1996, P. 324–341

2. Lattner C. and Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation // In Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, Palo Alto, USA, 2004, P. 85

3. Itanium C++ ABI:

   `http://refspecs.linuxbase.org/cxxabi-1.83.html`