О новом поколении промежуточных представлений, применяемых для анализа бинарного кода[1]

[1,2] *М.А. Соловьев <icee@ispras.ru>*
[1] *М.Г. Бакулин <bakulinm@ispras.ru>*
[1] *М.С. Горбачев <sadbear@ispras.ru>*
[1,2] *Д.В. Манушин <dman95@ispras.ru>*
[1,2] *В.А. Падарян <vartan@ispras.ru>*
[1] *С.С. Панасенко <spanasenko@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский государственный университет им. М.В. Ломоносова, 119991, Россия, г. Москва, Ленинские горы, д. 1*

Аннотация. Многие программные инструменты анализа бинарного кода работают не напрямую с машинными командами, а с промежуточными представлениями, в которые этот код транслируется. В статье рассмотрены различные задачи анализа бинарного кода, сформулированы требования к промежуточному представлению, которое могло бы использоваться сразу для многих задач. Базовые требования дополнены требованиями, вытекающими из особенностей целевых процессорных архитектур. Проведен обзор существующих подходов к декодированию машинных команд и описанию их семантики и предложены новые подходы к решению этих задач: унифицированная схема декодирования и промежуточное представление для задания семантики команд, позволяющее учитывать особенности выборки команд и обработки исключений. Показан способ применения предложенных подходов к моделированию работы процессора при конкретной и абстрактной интерпретации и символьном выполнении.

Ключевые слова: абстрактная интерпретация; анализ бинарного кода; динамический анализ; компиляторные технологии; обратная инженерия ПО; символьное выполнение; статический анализ.

DOI: 10.15514/ISPRAS-2018-30(6)-3

Для цитирования: Соловьев М.А., Бакулин М.Г., Горбачев М.С., Манушин Д.В., Падарян В.А., Панасенко С.С. О новом поколении промежуточных представлений, применяемом для анализа бинарного кода. Труды ИСП РАН, том 30, вып. 6, 2018 г., стр. 39-68. DOI: 10.15514/ISPRAS-2018-30(6)-3

[1] Работа поддержана грантом РФФИ 18-07-01256 А.

1. Введение

С задачами анализа исполняемого бинарного кода сталкиваются как разработчики ПО, так и специалисты по кибербезопасности. Необходимость проводить анализ на уровне бинарного кода обусловлена не только ситуациями, когда исходные тексты недоступны или частично утрачены. Для высокоуровневого языка программирования затруднительно либо невозможно оценить последствия срабатывания программного дефекта. Оптимизирующие преобразования кода, проводимые современными компиляторами, способны внести серьезные и потенциально эксплуатируемые дефекты, когда сталкиваются с определенными аспектами поведения программы, выходящими за рамки спецификации языка программирования [1]. Разработчики вынуждены работать с исполняемыми файлами при поиске дефектов методами динамического анализа, отладке и тестировании, в некоторых задачах повторного использования кода. Специалисты по аудиту безопасности оказываются в схожей ситуации при поиске недекларированных возможностей и оценке безопасности ПО.

В настоящий момент уже существует множество программных инструментов, в различной степени автоматизирующих анализ бинарного кода при решении практических задач. Общим местом у значительной части таких инструментов оказывается промежуточное представление, в которое переводится анализируемый код, подобно тому, как это происходит в компиляторе. Исторически такой способ работы с кодом применялся в бинарной трансляции, а затем – в бинарном инструментировании [2] и задачах анализа.

Распространенные процессорные архитектуры имеют достаточно обширные наборы команд, поэтому целесообразно реализовать сложный анализ на уровне промежуточного представления, упрощающего «крупные» команды целевой архитектуры путем их дробления на более мелкие элементы. Такой подход хорош также тем, что позволяет посредством трансляции в промежуточное представление кода различных целевых архитектур добиться более простой процедуры адаптации инструмента анализа к новой целевой архитектуре: достаточно разработать модуль трансляции.

Помимо положительных аспектов, практическое применение промежуточных представлений сталкивается с рядом проблем.

В первых работах [3] применялись промежуточные представления, непосредственно заимствованные из компилятора, а именно, LLVM [4]. К сожалению, такой подход оказался малоэффективен, т.к. существенная составляющая LLVM – типы данных – в бинарном коде отсутствует и не может быть восстановлена во время трансляции.

Следующая «волна» работ была посвящена разработке специализированных промежуточных представлений, отвечающих условиям анализа бинарного кода: Vine [5], Pivot [6], BAP [7] и REIL [8]. Эти представления изначально были рассчитаны на то, что будут получены из бинарного кода с целью анализа

свойств. Проблемы проявились в том, что разные программные инструменты применяют различные по своей форме и степени детальности промежуточные представления, а сам функционал трансляции каждый раз реализуется заново. Помимо очевидного увеличения трудозатрат на разработку новых инструментов, сложившаяся ситуация также выражается в возможности рассогласования результатов при использовании нескольких инструментов, так как разные инструменты могут иметь разные неточности в обработке отдельных команд и их характеристик.

Специализация представлений позволила относительно успешно применять их к коду процессорных архитектур общего назначения (x86, ARM, MIPS), но некоторые аспекты поведения этих машин и других, менее привычных (но не менее распространенных) процессорных архитектур эти представления описать не могут. В первую очередь это затрагивает особенности обработки исключений, прохождение команд по конвейеру и обращение в память. Вопросы безопасности устройств интернета вещей (IoT) делают все более острой необходимость анализировать код микроконтроллеров, тогда как возможностей имеющихся программных инструментов оказывается недостаточно.

В данной статье предлагается новое промежуточное представление, с одной стороны являющееся специализированным для решения задач анализа бинарного кода, а с другой – универсальным в своих возможностях менять уровень детализации при описании работы широкого класса процессорных архитектур, как общего назначения, так и применяемых в микроконтроллерах.

Дальнейший материал организован следующим образом. Во втором разделе рассматриваются наиболее распространенные задачи анализа бинарного кода с целью сбора требований к промежуточному представлению. В третьем разделе описаны ключевые особенности различных процессорных архитектур, которые также необходимо учесть при проектировании промежуточного представления. В четвертом разделе описаны существующие решения по декодированию машинного кода и предложен способ декодирования, который является составной частью разработанного подхода. Пятый раздел содержит обзор существующих подходов к моделированию операционной семантики бинарного кода и описывает предлагаемый в рамках данной работы подход. В шестом разделе приводится общая схема предлагаемого решения. Седьмой раздел содержит заключение и перечисление направлений дальнейших работ.

2. Типовые сценарии анализа бинарного кода

Для того чтобы определить требования к универсальному промежуточному представлению для анализа бинарного кода, необходимо рассмотреть распространенные задачи. На Рис. 1 такие задачи разнесены на разные уровни, исходя из того, решают они целевую задачу или играют вспомогательную роль. Условно можно сгруппировать их в три категории, согласно тому, какой

характер имеют действия над промежуточным представлением: абстрактная интерпретация, конкретное и символьное выполнение.

Рис. 1 – Задачи анализа бинарного кода, предполагающие использование промежуточного представления
Fig. 1 – Binary code analysis problems benefitting from use of an intermediate representation

2.1. Конкретное выполнение

Эмуляция на основе динамической двоичной трансляции (DBT) – традиционное и распространенное применение промежуточного представления, полученного из бинарного кода. Эмулируемый код линейно декодируется до передачи управления, полученный фрагмент кода (блок трансляции) переводится во внутреннее представление, над ним проводятся оптимизации, после чего представление транслируется в код хостовой машины [9]. Чтобы общая производительность эмулятора не страдала, необходимо добиться компромисса между качеством оптимизации и накладными расходами. В силу этого работа с промежуточным представлением ограничивается «легкими» оптимизациями: удалением мертвого кода, продвижением констант и копирований, анализом живых переменных.

В задачах автоматизации отладки, таких как отладка обращений к памяти, поиск гонок, и также при профилировании, трансляция дополняется инструментированием. Наиболее известны системы Valgrind [2], Pin [10] и DynamoRIO [11].

В целом, существуют два подхода к инструментированию [2]. При применении подхода “copy and annotate” проводится декодирование команд программы при ее выполнении и некоторые из них, в зависимости от решаемой задачи, заменяются вызовами вспомогательных функций, которые осуществляют дополнительную обработку, либо производится замена операндов и т.п. Так реализованы системы Pin и DynamoRIO. Второй подход, “disassemble and resynthesize”, схож с тем, как работает ранее описанная DBT с углубленным разбором операционной семантики каждой команды: инструментирование выполняется над промежуточным представлением, которое оптимизируется и используется затем для генерации кода хостовой машины. Этот подход,

реализованный в Valgrind, существенно более гибкий, но в то же время связан с повышенными накладными расходами.

От промежуточного представления в этих сценариях требуется быть удобным для проведения базовых оптимизаций: по возможности не иметь побочных эффектов и соответствовать SSA-форме; а также предоставлять простое, но емкое API к результатам декодирования команды: коду операции, операндам, режиму адресации и т.п.

2.2. Символьное выполнение

Символьное выполнение – более общая модель выполнения программы, по-прежнему ограниченная одним путем, но заменяющая конкретные значения переменных на абстрактные (символьные) значения. Как правило, формулы, задающие взаимосвязи между такими значениями, – это бесквантворные предикаты первого порядка над массивами битовых векторов (QF_ABV).

Символьное выполнение используется в задачах автоматического поиска программных дефектов и оценки их критичности. Архитектура соответствующих программных средств (S2E [12], Mayhem [13], работы А. Федотова и В. Каушана [14]) уже устоялась, она представляет собой цепочку нескольких видов анализа и преобразований кода (рис. 2).

Рис. 2 – Типовая архитектура инструмента символьного анализа
Fig. 2 – Generic architecture of a symbolic analysis tool

Выполнение программы, сопровождающееся поддержкой символьного состояния, обеспечивается средствами бинарного инструментирования (Valgrind [2], Pin [10], QEMU [9]). Динамический анализ помеченных данных отбирает команды, обрабатывающие пользовательский (символьный) ввод. Затем отобранные команды транслируются в промежуточное представление. Для промежуточного представления определяются правила символьной интерпретации, выполнение которых вырабатывает выражения над символьными переменными и константами. Для описания текущего пути выполнения все пройденные условные переходы транслируются в ограничения над символьными выражениями, что позволяет сформировать предикат пути. Предикат пути, дополненный предикатом безопасности – формальным описанием некоторого типа ошибки, предается в SMT-решатель.

Если операции промежуточного преставления оказываются «близки» к логике QF_ABV, то затраты на вторую трансляцию удастся уменьшить, а сами правила окажутся тривиальными.

Следует отметить, что возможна архитектура символьного анализа и с другим порядком действий, когда весь код транслируется в промежуточное представление и уже на нем ведется отслеживание символьных значений, без отдельного этапа отслеживания помеченных данных.

2.3. Абстрактная интерпретация

При проведении ручного анализа бинарного кода основным инструментом аналитика является дизассемблер. Дизассемблер осуществляет декодирование команд из сегмента кода одним из двух способов: либо последовательным просмотром от начала сегмента, либо методом рекурсивного спуска от точек входа [15]. В основном применяется второй способ, т.к. он сохраняет свою работоспособность в случае, когда между подпрограммами есть пропуски или данные (например, таблицы переходов). Также применяется подход спекулятивного дизассемблирования [16], когда каждое смещение в сегменте кода рассматривается как возможное начало кода функции, а затем на этапе разрешения конфликтов отбрасываются неверно выбранные.

Таким образом, для решения задачи дизассемблирования требуется, как минимум, декодер машинных команд с представлением результата в ассемблерном виде, а для подходов, основанных на рекурсивном спуске, необходима также классификация команд по признаку передачи управления и возможность вычисления целевого адреса перехода. Один из способов, как можно вычислить возможные исполнительные адреса – применение алгоритма VSA [17], способного аппроксимировать значения адресов и целых чисел. Результат его работы позволяет не только обнаружить фрагменты кода (по адресам перехода), но и улучшить результаты при поиске дефектов и уязвимостей путем статического анализа.

Для поиска дефектов и уязвимостей в бинарном коде применяются подходы, основанные на статическом анализе и символьном выполнении. Например, система BINSIDE [18] использует промежуточное представление REIL [8] для последующей трансляции в бит-код LLVM [4], поверх которого при помощи разработанных чекеров ведется поиск определенных видов дефектов.

По своей сути и статический анализ, и символьное выполнение представляют собой разновидности абстрактной интерпретации [19]. Разница заключается в том, что в первом случае одновременно исследуются все пути в программе (и, как правило, при помощи итеративного применения передаточных функций находится некоторое решение, соответствующее неподвижной точке – MFP-решение), а во втором случае исследуется некоторое подмножество путей, причем каждый путь анализируется независимо. В случае, когда удается рассмотреть все возможные пути, полученное решение (MOP-решение)

оказывается более точным, чем MFP-решение. Однако на практике просмотр всех путей возможен только для самых примитивных программ.

Вне зависимости от того, каким образом применяется абстрактная интерпретация, от инфраструктуры анализа требуется декодирование команд и представление их семантики в упрощенном виде для того, чтобы решетки и передаточные функции, которые задают абстрактную интерпретацию, были менее сложными в описании. Именно поэтому в задачах поиска дефектов и уязвимостей в бинарном коде чаще всего можно встретить использование достаточно мощных промежуточных представлений. Кроме системы BINSIDE, схожие задачи решаются в системах BitBlaze [5] и BAP [7], каждая из которых имеет свое промежуточное представление для описания семантики машинных команд.

3. Особенности целевых процессорных архитектур

Как показывает практика авторов при работе над задачами динамического анализа бинарного кода [20], все процессорные архитектуры общего назначения достаточно близки друг к другу как с точки зрения наборов команд и их кодировки, так и с точки зрения функционирования конвейера и подсистемы памяти. Существенные отличия от процессоров общего назначения проявляют микроконтроллеры и DSP-процессоры. Среди таких отличий – принадлежность к «не-фон-Неймановским» архитектурам, разрывное размещение команд в памяти, отсутствие останова конвейера при конфликте по данным или управлению и т.п. В данном разделе приведен обзор наиболее важных особенностей некоторых целевых процессорных архитектур, которые должны быть учтены при разработке промежуточного представления для анализа бинарного кода.

3.1. Семейство ARM

Семейство процессорных архитектур ARM в настоящее время представлено архитектурами ARMv7 и ARMv8, которые в свою очередь имеют деление на профили “A”, “R” и “M”. В зависимости от модели, процессор ARM может поддерживать до трех наборов команд: наборы A32 и A64 с фиксированной длиной команд и набор T32 с переменной длиной команд. В каждый момент времени текущий используемый набор команд определяется управляющими регистрами машины и может быть изменен во время работы при помощи специальных команд.

Среди особенностей можно выделить то, каким образом реализовано условное выполнение команд. В наборе A32 практически все команды имеют поле cond, определяющее, при каких условиях команда будет выполняться. В наборе A64 только небольшая часть команд имеет такое поле. В наборе T32, помимо нескольких команд, которые имеют явное поле условия, имеется команда IT,

которая в зависимости от вычисленного значения условия определяет, какие из команд следующего условного блока (4 команды) будут выполнены при истинном значении, а какие – при ложном.

Другой особенностью является наличие сопроцессоров, работа с которыми ведется при помощи команд MCR и MRC. Специализированные решения на базе архитектуры ARM могут иметь помимо стандартных сопроцессоров произвольные расширения.

3.2. Семейство AVR

Семейство микроконтроллеров AVR имеет очень небольшой набор команд. Команды могут иметь длину в 16 или 32 бита, длина команды может быть однозначно определена по битовому префиксу. Микроконтроллеры AVR имеют отдельное пространство памяти кода и памяти данных, причем часть последнего используется для доступа к периферии. Таким образом, основной особенностью является принадлежность этих микроконтроллеров к «не-фон-Неймановским» архитектурам.

3.3. Семейство MIPS

Семейство процессорных архитектур MIPS является характерным примером RISC-архитектуры: команды имеют фиксированную длину с простой структурой и имеют достаточно простую семантику, благодаря чему легко могут быть декодированы и проанализированы с точки зрения их поведения. Однако в силу того, что существует большое число версий архитектуры MIPS и множество различных расширений, без точного знания версии и набора реализованных расширений некоторые команды не могут быть декодированы однозначно. Номер версии архитектуры может быть считан из системных регистров (при этом некоторые эмуляторы не вполне точно реализуют этот функционал), но набор расширений, поддерживаемый данной машиной, необходимо получать извне.

Конвейер MIPS работает со слотами задержки, обработка команд передачи управления должна осуществляться с учетом этой особенности.

3.4. Семейство PowerPC

Процессорные архитектуры семейства PowerPC также являются яркими представителями RISC-архитектур: все команды имеют фиксированную длину с небольшим количеством форматов кодировки (т.е. расположения полей). Семантика команд несложна, и единственной отличительной особенностью является возможность явного указания при арифметических командах флага, включающего или выключающего обновление слова состояния в регистре *XER*.

3.5. Семейство RISC-V

Новое семейство процессорных архитектур RISC-V было спроектировано таким образом, чтобы максимально упростить реализацию «в железе» процессоров этой архитектуры и инструментов, которые работают с бинарным кодом RISC-V. Команды имеют равную длину, имеется четкое разбиение на подмножества команд (каждая реализация может поддерживать только часть из них), причем принадлежность команды к определенному подмножеству может быть определена по битовой маске. Большая часть подмножеств работает всего с четырьмя форматами кодировок команд, что делает декодирование несложным. Однако имеется одно исключение: расширение “C” набора команд дополняет «обычные» 32-разрядные кодировки команд 16-разрядными кодировками, которые позволяют кодировать некоторые команды базового набора более кратко. Длина команды всегда определяется двумя битами в кодировке.

3.6. Семейство Texas Instruments

Семейство процессорных архитектур Texas Instruments (TMS) представлено обширным набором микроконтроллеров. Наиболее яркие особенности имеют микроконтроллеры с явным параллелизмом на уровне команд, например TMS320C67x. Здесь единицей выборки является не одна команда, а пакет, состоящий из восьми команд. Кодировка пакета указывает, какие из этих команд могут быть выполнены параллельно. Интересным моментом является то, что процессор допускает передачу управления в середину пакета, что соответствует частичному его выполнению.

Все процессорные архитектуры TMS имеют слоты задержки, соответственно, команды передачи управления требуют специализированной обработки.

3.7. Семейство x86

Процессорные архитектуры семейства x86 (в том числе x86-64) в настоящее время наиболее распространены среди процессорных архитектур общего назначения и, тем самым, бинарный код для этих архитектур чаще всего становится объектом анализа. Семейство x86 относится к CISC-архитектурам, имеет огромное число команд, множество режимов адресации, несколько вариантов кодировки одних и тех же команд (например, с использованием VEX-префикса или без него), переменную длину команды, которая не может быть вычислена по какому-либо префиксу фиксированного размера, и т.п. Кроме того, многие команды (особенно системные) весьма сложны, что требует их упрощения при анализе путем разбиения на более мелкие составные части. Таким образом, полноценная поддержка целевой архитектуры x86 является непростой задачей с точки зрения реализации декодера команд и описания их операционной семантики.

С другой стороны, конвейер x86-процессоров с точки зрения разработчика достаточно прост и не имеют каких-либо существенных особенностей, влияющих на анализ кода. То же можно сказать и про систему памяти: ее сложность заключается лишь в количестве последовательных трансляций адресов, но каждый отдельный этап не является сложным. Определенным исключением из сказанного являются новые наборы команд, связанные с организацией транзакционной модели памяти (TSX) и защищенных областей памяти (SGX), однако в зависимости от решаемой задачи их обработка часто может быть существенно упрощена без потери точности результатов.

3.8. Семейство Xtensa

Семейство микроконтроллеров Xtensa достаточно популярно при реализации устройств интернета вещей, поскольку решения на базе Xtensa предоставляют доступ к 802.11-сетям (WiFi) и Bluetooth «из коробки». Базовый набор команд очень невелик (менее 100 простых команд), команды имеют фиксированную длину в 24 бита и простую кодировку. Подобно тому, как устроено расширение “C” в RISC-V, некоторые микроконтроллеры Xtensa поддерживают короткую запись длиной 16 бит для части команд. В некоторых моделях поддерживаются аппаратные циклы: регистр *LBEG* определяет адрес начала цикла, регистр *LEND* – адрес конца, а регистр *LCOUNT* содержит число оставшихся операций. При выборке команд процессор автоматически осуществляет декремент значения *LCOUNT* при достижении адреса *LEND* и при необходимости передает управление на адрес *LBEG*. Это поведение необходимо учесть при анализе потока управления кода для Xtensa.

Процессоры семейства Xtensa также могут работать в связке с внешними сопроцессорами, поэтому для полноценного анализа бинарного кода этой целевой архитектуры необходимо знать, какие команды добавляют к базовому набору эти сопроцессоры.

3.9. Требования к промежуточному представлению

Подводя итоги перечисленных выше особенностей, можно сформулировать набор требований, которым должно удовлетворять представление для анализа бинарного кода, чтобы оно было пригодно для разностороннего анализа всех перечисленных процессорных архитектур.

I. Требуется поддержка «не-фон-Неймановских» архитектур, имеющих несколько адресных пространств памяти.
II. Некоторые архитектуры поддерживают аппаратные циклы и другие механизмы, влияющие на выборку команд. Как следствие, наличие таких механизмов необходимо учитывать при интерпретации и анализе потока управления.

III. Многие RISC-архитектуры имеют слоты задержки, причем их число и поведение команд передачи управления с точки зрения выполнения или отбрасывания команд в слотах задержки в разных архитектурах может отличаться. Такая информация тоже влияет как на интерпретацию, так и на анализ потока управления.

IV. Необходимо поддержать возможность работы с кодировками команд переменной длины.

V. Во многих процессорных архитектурах декодирование команды зависит от значений управляющих регистров (например, текущий набор команд в ARM, признак разрядности кода в x86 и т.п.). Декодер должен иметь доступ к контексту, откуда могут быть получены эти сведения.

VI. Некоторые процессорные архитектуры имеют различные версии и наборы расширений. Эта информация также должна использоваться при декодировании, т.к. иначе однозначное декодирование команд невозможно.

VII. При декодировании необходима поддержка процессорных архитектур с явным параллелизмом на уровне команд.

4. Декодирование машинных команд

Любое промежуточное представление для анализа бинарного кода стоится в два этапа, первым из которых является декодирование команд анализируемой программы или системы, а вторым собственно трансляция их в некоторую единообразную форму, которая в дальнейшем и анализируется. В данном разделе обсуждается первый из этих этапов с учетом тех требований, которые были сформулированы выше в подразделе 3.9.

4.1. Обзор существующих решений

Среди существующих программных средств, которые могут использоваться для декодирования бинарного кода, можно выделить две основные группы. К первой группе относятся декодеры, которые обеспечивают поддержку какой-либо одной целевой архитектуры (или семейства). Такие средства имеют, как правило, достаточно детальное представление разобранной команды: не только текстовый вид команды, но и ее структура с точки зрения мнемоники, префиксов, флагов, режимов адресации операндов и т.п. В то же время тот факт, что для различных целевых архитектур нужно использовать различные библиотеки декодирования с различными программными интерфейсами означает, что в инструментах, настраиваемых по целевой процессорной архитектуре, потребуется реализация дополнительного уровня унификации этих интерфейсов, что может нивелировать выгоду от использования готовой библиотеки.

Вторая группа представлена средствами, поддерживающими множество целевых процессорных архитектур и формирующими результат в виде универсальных структур, не зависящих от текущей архитектуры. На данный момент наиболее часто встречается использование библиотек из состава binutils [21], библиотеки Capstone [22] и декодирование команд средствами интерактивного дизассемблера IDA Pro [23]. Рассмотрим возможности, предоставляемые этими средствами, более подробно.

Библиотека декодирования из состава binutils в основном используется для получения текстового представления декодированной команды. Логика декодирования описана в виде функций на языке Си без использования внешних спецификаций и чаще всего представлена в виде каскада операторов *switch*, что делает отладку декодеров и внесение изменений затруднительным. Основным достоинством binutils является обширный набор поддерживаемых целевых архитектур, однако качество такой поддержки плавает: для основных архитектур общего назначения декодеры хорошо отлажены, но качество кода для декодирования команд микроконтроллеров существенно ниже.

Библиотека Capstone позволяет получать не только текстовый вид команды, но и структуру, где единообразно (т.е. в независимом от целевой архитектуры виде) описаны ее основные свойства. Среди таких свойств принадлежность команды к определенной группе (например, к командам передачи управления) и список читаемых и записываемых командой регистров. Поддерживается несколько целевых архитектур, в т.ч. ARM, M68K, MIPS, PowerPC, TMS320C64x, x86 и др. В то же время библиотека имеет ряд архитектурных ограничений, которые не позволяет с ее помощью решить все задачи, связанные с декодированием машинных команд. Некоторые наиболее существенные ограничения перечислены далее.

I. Детальная информация о команде доступна для каждой целевой архитектуры в виде своей структуры данных, что существенно уменьшает степень универсальности библиотеки с точки зрения интерпретации результатов декодирования.

II. Даже в рамках структур с детальной информацией отсутствуют сведения о количество и способе обработки слотов задержки, а такая информация существенна для всех потоково-чувствительных видов анализа бинарного кода.

III. Отсутствует поддержка архитектур с явным параллелизмом на уровне команд.

IV. Нет возможности получить список участков памяти, с которыми работает команда. Таким образом, анализ потока данных при использовании этой библиотеки ограничен регистрами.

Реализация декодеров в Capstone выполнена в виде кода на языке Си с обширным применением макросов, что делает поддержку кода непростой задачей.

Таким образом, библиотека Capstone хорошо подходит для поверхностного анализа бинарного кода (в частности, задач дизассемблирования и простых случаев анализа потоков управления и данных), но не пригодна в текущем виде для углубленного анализа.

Интерактивный дизассемблер IDA Pro является в настоящий момент инструментом выбора при автоматизированном исследовании бинарного кода, когда пользователь работает с восстановленным по бинарному кода ассемблерным листингом. Дизассемблер поддерживает множество целевых архитектур и имеет SDK для реализации такой поддержки в виде подключаемых модулей. Однако качество существующих декодеров сильно отличается от модуля к модулю. В частности, информация о читаемых и записываемых командой регистрах доступна лишь для части целевых архитектур, в другой части она не реализована или работает некорректно. Что касается предлагаемого SDK, его длинная история и тот факт, что изначально интерфейсы IDA Pro проектировались только для кода архитектуры x86, привели к тому, что программные интерфейсы являются достаточно запутанными, имеют множество неявных контрактов и т.п.

Каждый модуль декодирования реализуется на языке Си++ или Python и, так же, как и в перечисленных выше средствах, логика декодирования описана прямо в коде. Единого способа декларативного описания набора команд не предусмотрено.

Таким образом, перечисленные средства декодирования, являясь отличными решениями для определенных узких задач, не являются универсальными. В следующем подразделе предлагается подход к декодированию, основанный на декларативном задании набора команд, который является более гибким и, по мнению авторов, легче поддерживаемым.

4.2. Предлагаемый подход к декодированию

При разработке предлагаемого подхода к декодированию помимо тех требований, которые вытекают из особенностей процессорных архитектур, в качестве основополагающего также выдвигалось требование получения унифицированного декодера, т.е. такого который обрабатывает декларативно заданную спецификацию конкретной архитектуры набора команд (ISA) и предоставляет для всех архитектур одинаковый программный интерфейс.

Команда	Кодировка (младшие биты справа)	Предикат
lw x8, 0(x9)	000000000000_01001_010_01000_0000011	
lw x9, 1(x8)	000000000001_01000_010_01001_0000011	
ld x8, 0(x9)	000000000000_01001_011_01000_0000011	поддержка команд RV64I
or x6, x5, x4	0000000_00100_00101_110_00110_0110011	
ori x6, x5, 4	000000000100_00101_110_00110_0010011	

Рис. 3. Фрагмент таблицы декодирования для RISC-V
Fig. 3. RISC-V decode table fragment

Вообще говоря, с логической точки зрения спецификация кодировок команд некоторого набора может быть описана в виде таблицы, которая ставит в соответствие каждой конкретной команде полную кодировку (т.е. такую, где все биты имеют конкретные значения) и предикат, накладываемый на текущее состояние машины. Фрагмент такой таблицы для процессоров RISC-V приведен на рис. 3. Предикат необходим, чтобы учесть переключаемые во время выполнения наборы команд (например, в процессорах ARM), размеры операндов по умолчанию (например, в процессорах x86), состав расширений архитектуры и другие подобные характеристики, не присутствующие явно в кодировке команды. Понятно, что подобная таблица будет иметь огромный размер, поэтому при проектировании логики декодирования основной целью является структуризация этой таблицы путем группировки ее строк. Если временно отбросить необходимость работы с предикатом, то остается набор битовых строк, которым соответствуют различные команды. Структуры данных для быстрого поиска в таких массивах данных хорошо известны, одной из них является сжатый бор, который группирует битовые строки по общим префиксам (рис. 4). Однако в реальности весьма часто встречаются ситуации, когда определить, например, мнемонику команды по нетривиальному префиксу невозможно. В частности, в архитектуре RISC-V многие команды имеют разбитое на две части поле кода операции: часть битов располагается в начале, а часть – в самом конце кодировки. Таким образом, требуется модификация структуры бора для адаптации к решаемой задаче.

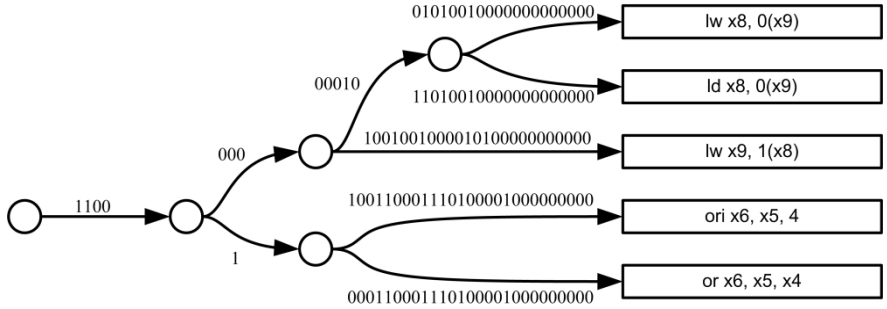

Рис. 4. Фрагмент сжатого бора для RISC-V
Fig. 4. RISC-V compressed trie fragment

Заметим, что можно рассматривать отдельные битовые поля из кодировок команд не обязательно последовательно. Если разбить кодировку команды на отдельные поля, то, вообще говоря, можно проводить сравнения битов в этих полях с записанными в таблице в произвольном порядке. Такой подход может быть формализован в виде автомата и соответствующего ему дерева, где на ребрах указаны значения битовых полей, совпадение с которыми разрешает переход в соответствующую вершину. Еще раз отметим, что порядок ребер

вдоль пути от корня дерева к листу не имеет значения, т.е. такие деревья можно преобразовывать путем перестановки вершин, не меняя задаваемые ими кодировки.

Структура кодировок команд в процессорах, как правило, такова, что сначала может быть декодирована некоторая общая форма команды (код операции вместе с модификаторами, набор режимов адресации и размеры операндов), а далее определены и сами операнды (т.е. конкретные регистры, адреса памяти и т.п.). Такая структура объясняется тем, как обычно «в железе» реализуются начальные этапы конвейера. Исходя из этих соображений, разобьем дерево на два слоя. В верхнем (ближе к корню) слое сосредоточим проверку битовых полей, определяющих код операции и режимы адресации операндов, а в нижнем (ближе к листьям) – связанные с окончательным определением операндов. Если построить такое двухслойное дерево, то можно будет заметить, что нижний уровень будет содержать большое количество повторяющихся шаблонов. Перейдя от дерева к ациклическому графу, эти повторы можно ликвидировать. Однако при этом только часть результата декодирования может храниться в листьях, а часть должна накапливаться в процессе прохода по верхнему слою. Пример такого графа показан на Рис. 5, для того, чтобы не перегружать иллюстрацию, ребра между слоями не показаны. В левой части рисунка находится слой декодирования кода операции и режимов адресации операндов, в правой части – слой декодирования самих операндов. При этом набор операндов, которые будут декодироваться, определяется листовой вершиной первого слоя.

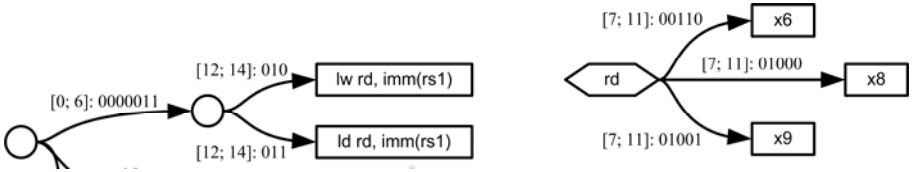

Рис. 5. Фрагмент ациклического графа декодера RISC-V
Fig. 5. RISC-V decode acyclic graph fragment

Вернемся теперь к влиянию состояния машины на декодирование команд. Заметим, что, по сути, соответствующий предикат фигурирует как конъюнкт в общем виде кодировки наряду с конъюнктами, отвечающими за проверку значений битовых полей. Тогда такие предикаты могут быть добавлены в рассматриваемый граф в виде еще одного слоя, наиболее близкого к корню. Правило перехода по ребрам в этом слое следующее: переход возможен, если указанный на ребре предикат на значения частей системных регистров и характеристики конкретной модели процессора принимает истинное значение.

Если описанный граф для какого-либо процессора построен, то результатом декодирования команды является совокупность информации в вершинах, отвечающих предикату и битовой кодировке команды или, говоря иначе, путь

на этом графе. С частями графа, описывающими декодирование операндов, можно связать соответствующие режимы адресации. Для каждого режима адресации можно определить набор характеристик операнда, которые в рамках этого режима не являются фиксированными. В примере для RISC-V для режима адресации «регистр» такой характеристикой будет номер регистра, для режима «константа» – номер константы, а для режима «косвенная адресация» – номер регистра и смещение. Тогда часть пути в графе, проходящая по слою кода операции, задает мнемонику и режимы адресации операндов, а часть, проходящая по слою операндов, окончательно декодирует характеристики операндов. Например, результатом декодирования битового вектора 000000000000_01001_010_01000_0000011 будет структура, содержащая следующие сведения:

- мнемоника команды: LW;
- первый операнд имеет режим адресации «регистр», характеристика «номер регистра» имеет значение “x8”;
- второй операнд имеет режим адресации «косвенная адресация»;
- характеристика «номер регистра» имеет значение “x9”, а «смещение» равно 0.

Такое представление результатов декодирования позволяет:

- построить текстовое представление команды;
- при наличии сохраненного пути по графу восстановить битовую кодировку;
- иметь возможность определять в унифицированном виде формулы, отбирающие какие-либо команды по мнемонике и набору операндов, что будет необходимо в дальнейшем при трансляции отдельных машинных команд в промежуточное представление, описывающее операционную семантику.

5. Описание операционной семантики машинных команд

В предыдущем разделе были рассмотрены вопросы декодирования. Перейдем теперь к обсуждению вида той части промежуточного представления, которая отвечает за описание операционной семантики машинных команд.

5.1. Обзор существующих решений

Как было сказано во введении, появление первых промежуточных представлений, достаточно точно описывающих операционную семантику бинарного кода, обязано в первую очередь задачам динамической бинарной трансляции и инструментирования. В частности, полносистемный эмулятор с открытым исходным кодом QEMU [9] основан на бинарной трансляции гостевого кода в код машины, на которой производится эмуляция. Трансляция работает на базе промежуточного представления TCG. Гостевой код

переводится в это представление, а затем по нему осуществляется генерация кода инструментальной машины. Аналогичный подход реализован в системе динамического бинарного инструментирования Valgrind [2], где применяется промежуточное представление VEX. Представления TCG и VEX достаточно близки по своим основным характеристикам и без существенных изменений много лет успешно применяются в своих задачах.

Однако для решения задач углубленного анализа потоков данных и управления в бинарном коде эти представления малопригодны: и TCG, и VEX имеют поддержку вызова вспомогательных функций на языке Си (helpers), которые могут произвольным образом менять состояние анализируемой системы. Эти функции встречаются достаточно часто, поскольку используются для эмуляции сложных команд целевых процессоров.

Еще одной проблемой данных представлений является отложенная обработка флагов, которая оправдана при динамическом анализе (т.к. позволяет не высчитывать значения флагов, которые в дальнейшем не будут использоваться), но существенно затрудняет статический анализ и символьное выполнение на базе такого представления, поскольку такое поведение разрывает цепочки зависимостей по данным. Кроме того, в обеих системах этапы построения промежуточного представления и кодогенерации описаны в виде модулей на языке Си, что затрудняет их модификацию и отладку.

Специализированные промежуточные представления для анализа бинарного кода, лежащие в основе таких систем, как Vine [5], BAP [7], а также представление REIL [8] не имеют неявных действий, поэтому более пригодны для различных сценариев анализа, хотя и более «многословны». Трансляторы здесь так же, как и в QEMU и Valgrind, реализованы в виде программных модулей и не поддерживают декларативное задание набора команд целевой машины. Кроме того, в этих представлениях фиксировано множество элементарных операций, на которые требуется разбивать поведение машинной команды, причем это множество ограничено: в своем текущем виде эти представления пригодны, в основном, для команд целочисленной арифметики, побитовой логики и передачи управления. В этих представлениях также нет моделей памяти и конвейера, обработки исключений, что делает их использование в задачах полносистемного анализа бинарного кода затруднительным.

Представление Pivot [6] было предложено в первую очередь для решения задач динамического оффлайн-анализа бинарного кода процессорных архитектур x86 (и x86-64), PowerPC, MIPS и ARMv6, однако впоследствии было также успешно применено для некоторых сценариев статического анализа бинарного кода этих архитектур. Ключевыми отличиями этого представления являются:

- произвольный набор адресных пространств (в том числе пространств регистров), что позволяет поддерживать «не-фон-Неймановские» архитектуры и отделить регистры целевой машины от временных переменных (виртуальных регистров);
- временные переменные находятся в форме статического единичного присваивания, что упрощает дальнейший статический анализ;
- набор базовых операций, через которые выражается операционная семантика, не фиксирован: в зависимости от задачи пользователь может либо описать новые операции, либо выразить семантику команды через существующие;
- операции могут иметь неявные побочные эффекты, выражающиеся в чтении или записи отдельных битов слова состояния, но для каждой операции эти множества специфицированы, что является некоторым компромиссом между полностью явным описанием всех эффектов и отложенным вычислением флагов;
- реализован язык задания внешних спецификаций семантики команд, трансляторы строятся автоматически по этим спецификациям.

Несмотря на успешный опыт использования данного представления в ряде проектов ИСП РАН, со временем проявились некоторые его принципиальные ограничения. Во-первых, при его разработке не была учтена потребность в описании поведения машины «между командами», т.е. при выборе очередной команды, обработке прерываний и т.п.

Во-вторых, простая модель памяти различных адресных пространств (байтовые массивы), используемая в Pivot, в полносистемных задачах анализа оказывается недостаточной: необходимо учитывать такие особенности функционирования аппаратуры, как трансляция адресов (в том числе многоуровневая), отображенные на память устройства и прямой доступ к памяти (DMA).

В-третьих, набор библиотек, которые используются при работе с эти представлением, включает только средства трансляции и конкретной интерпретации, а всю логику последующего анализа (например, итеративный алгоритм для поиска MFP-решения задачи потока данных или логику продвижения символьного состояния) приходится реализовывать в каждом случае заново.

Наконец, среди академических работ, посвященных вопросам описания операционной семантики машинных команд, можно выделить различные диалекты языка nML [24], языки ISDL [25], L3 [26] и SAIL [27]. Наиболее интересен подход, применяемый в L3 и SAIL: здесь для описания и декодирования, и поведения команд используется диалект чисто функционального языка. Вся работа машины, по сути, описана одной чистой функцией, которая принимает на вход текущее состояние и кодировку очередной команды и возвращает новое состояние машины. Таким образом, работа машины по выполнению некоторой последовательности команд может быть описана как композиция вызовов этой функции. Код чисто функциональной парадигмы может быть проанализирован гораздо легче, чем

машинный код, поэтому применяемый функциональный язык и используется в качестве промежуточного представления.

К недостаткам здесь можно отнести ограниченную выразительную мощность перечисленных языков (в частности, имеются сложности с описанием команд, работающих с числами с плавающей точкой, векторных команд, системных команд), отсутствие возможности задания особенностей выборки команд (слоты задержки, аппаратная поддержка циклов) и обработки прерываний и исключений, т.е. действий по аппаратному переключению контекста. Кроме того, отсутствует поддержка «не-фон-Неймановских» архитектур.

5.2. Предлагаемое промежуточное представление

В настоящем подразделе описано предлагаемое промежуточное представление Pivot 2, являющееся развитием разработанного авторами ранее представления, описанного в работе [6].

5.2.1. Адресные пространства

Представление Pivot 2 единообразно описывает все данные, с которыми могут работать команды целевого процессора, в виде набора *адресных пространств*. Каждое адресное пространство считается изолированным и имеет свой фиксированный размер адреса. Одним из адресных пространств является пространство регистров: регистрам целевой машины назначаются некоторые адреса с учетом их вложенности, и в дальнейшем работа с этим пространством ведется так же, как и с пространством памяти. Для «не-фон-Неймановских» архитектур такая модель позволяет описать все используемые пространства памяти, не прибегая к трансляции адресов и другим подобным приемам.

Адресные пространства делятся на два типа: *локальные* и *удаленные*. Под локальными адресными пространствами понимаются такие, которые могут быть рассмотрены как простой массив байтов, локальный для анализируемого потока выполнения. В частности, это означает, что значение, записанное по какому-либо адресу, при последующем чтении будет получено без изменений. Кроме того, доступ к локальным адресным пространствам всегда должен заканчиваться успешно, он не может приводить к исключению. Примером локального адресного пространства является набор регистров общего назначения.

Удаленные адресные пространства, во-первых, не гарантируют какого-либо конкретного поведения при доступе (например, пространство портов ввода-вывода или пространство памяти с отображенными на память устройствами); и, во-вторых, могут заканчиваться ошибкой. Для удаленных адресных пространств помимо размера адреса задается размер битового вектора, описывающего возникшую ошибку. Такое разделение позволяет моделировать поведение команд процессора, которые могут вызывать исключения.

При работе с промежуточным представлением все адресные пространства, с которыми ведется работа, объединяются в *таблицу адресных пространств*, которая сопоставляет с каждым пространством некоторый индекс.

5.2.2. Операции

Наиболее мелкой единицей задания операционной семантики в представлении Pivot 2 является *операция*. Операция принимает на вход набор битовых векторов и формирует другой набор битовых векторов на выходе, т.е. может иметь произвольное число аргументов и результатов. Набор операций не фиксирован, что предоставляет возможность относительно несложного расширения промежуточного представления. Вместе с тем, к операциям предъявляются два следующих важных требования:

- операция не имеет каких-либо входов и выходов, кроме своих формальных аргументов и результатов, т.е. является «чистой» функцией;
- в рамках одной операции зависимости по данным представляют собой полный двудольный граф: все входы влияют на все выходы.

Таким образом, некоторая функция *add*, которая принимает два 32-разрядных числа и возвращает их сумму и признак переполнения, является операцией, т.к. на каждый из вырабатываемых результатов влияют все аргументы. Функция *xchg*, которая принимает два битовых вектора некоторой длины и возвращает их в измененном порядке, не является операцией, т.к. нет влияния каждого входа на каждый выход.

Сформулированные свойства операций позволяют проводить некоторые виды анализа потока данных даже без знания семантики каждой операции, например построение срезов или анализ помеченных данных.

Все используемые при анализе операции объединяются в *таблицу операций*, где за каждой операцией закрепляется некоторый индекс.

5.2.3. Временные переменные

Так как Pivot 2 является представлением бинарного кода, оно не имеет типов, и работает только с битовыми векторами различной длины. Все временные переменные находятся в форме статического единичного присваивания, тем самым размер переменной определяется один раз в точке ее определения. Временные переменные имеют нумерацию, начиная с 1. Эти номера локальны в пределах фрагмента, структурного элемента представления, который описан далее в п. 5.2.6. Временная переменная с номером 0 воспринимается особым образом: она считается всегда определенной как битовый вектор из единственного нулевого бита.

Набор временных переменных с последовательными номерами образует *группу*, которая может быть задана как полуинтервал номеров входящих в нее переменных. Понятие группы будет использоваться в дальнейшем при обсуждении операторов и передачи данных между базовыми блоками.

5.2.4. Операторы

Оператор представляет собой абстракцию одного неделимого действия в рамках промежуточного представления. Набор операторов в предлагаемом представлении фиксирован и не зависит от специфики набора команд целевой машины. Операторы могут принимать на вход набор номеров временных переменных (возможно пустой) и формировать на выходе группу временных переменных (также возможно пустую). В отличие от операций, некоторые операторы имеют побочные эффекты, но их суть фиксирована для каждого типа оператора. Операторы либо осуществляют пересылку данных (**MIX**, **EXTRACT**, **CONCAT**, **INIT**), либо выполняют некоторые действия (**INVOKE** и **CALL**), либо описывают обращения к запоминающим устройствам целевой машины (**LOAD.L**, **LOAD.R**, **STORE.L** и **STORE.R**).

Оператор MIX используется для группировки переменных: принимаемые на вход временные переменные последовательно копируются в переменные выходной группы. Например, оператор **MIX**, имеющий на входе набор { 1, 7, 2 }, а на выходе формирующий группу, начиная с номера 10, осуществит копирование переменной 1 и переменную 10, переменной 7 в переменную 11 и переменной 2 в переменную 12. В граничном случае, когда обрабатываемый набор переменных пуст, оператор **MIX** вырождается в пустой оператор **NOP**.

Оператор EXTRACT используется для выделения части битового вектора в своей единственной входной переменной в отдельную выходную переменную. При этом полуинтервал индексов битов, ограничивающий выделяемое поле, является константным параметром оператора. Индексы не могут зависеть от временных переменных. Данный оператор позволяет проводить более точный анализ помеченных данных даже в тех случаях, когда не учитывается семантика отдельных операций.

Оператор CONCAT конкатенирует несколько битовых векторов в один результирующий. Размер формируемой переменной определяется как сумма размеров входных переменных.

Оператор INIT заносит в переменную константный битовый вектор.

Оператор INVOKE применяет указанную (заданную индексом в таблице операций) операцию к набору переменных и формирует на выходе группу, соответствующую результату операции.

Оператор CALL вызывает фрагмент промежуточного представления (заданный индексом в таблице фрагментов) как подпрограмму. Подобно оператору **INVOKE** он принимает набор входных переменных и формирует на выходе группу. При интерпретации поддерживается стек вызовов, элементами которого являются пары (*индекс вызывающего базового блока*, *номер оператора в блоке*). Таким образом, при достижении выходной вершины вызванного фрагмента управление будет возвращено на следующий оператор после оператора **CALL.**

Оператор LOAD.L описывает чтение из локального адресного пространства. Для таких операторов указывает индекс адресного пространства из таблицы, номер переменной, содержащей адрес и размер считываемого значения. Также указывается порядок байтов: little endian или big endian. Это необходимо из-за того, что даже в рамках одного набора команд могут встречаться команды, которые по-разному воспринимают порядок байтов в памяти. Результат чтения заносится в указанную временную переменную.

Оператор STORE.L описывает запись в локальное адресное пространство: указывается индекс адресного пространства, номер переменной с адресом и номер переменной с записываемым значением, а также порядок байтов.

Оператор LOAD.R используется для осуществления чтения из удаленного адресного пространства. Напомним, что чтение из удаленного адресного пространства может закончиться неуспешно, поэтому помимо тех атрибутов, которые задаются для оператора **LOAD.L**, в данном случае дается также ссылка на фрагмент-обработчик ошибки. Этому фрагменту в качестве входного параметра будет передаваться битовый вектор с описателем ошибки, размер которого задан для каждого удаленного адресного пространства.

Оператор STORE.R осуществляет запись в удаленное адресное пространство и отличается от **STORE.L** тем же, чем **LOAD.R** от **LOAD.L**.

Операторы **LOAD.R** и **STORE.R** могут быть аннотированы флагом **PROBE**, который меняет их поведение следующим образом: само обращение не производится, а лишь проверяется его допустимость. Если обращение может быть проведено успешно, то выполнение переходит к следующему оператора, а иначе передается на обработчик ошибки.

Таким образом, всего зафиксировано 10 операторов, каждый из которых имеет понятное поведение и достаточно легко может быть проанализирован, что являлось одной из целей при проектировании промежуточного представления.

5.2.5. Базовые блоки

Неразрывная с точки зрения потока управления последовательность операторов объединяется в *базовый блок*. Помимо этой последовательности, базовый блок указывает также *входную группу* локальных переменных и *выходную группу*. Когда управление передается из одного базового блока в другой, значения переменных из выходной группы первого из них копируются в переменные входной группы второго. Таким образом, если два блока соединены ребром, они должны быть согласованы по входу и выходу. Такой подход позволяет не вводить в явном виде φ-функцию, заменив ее простыми действиями, производимыми на ребрах.

Для упрощения работы с промежуточным представлением любой базовый блок имеет формально два исходящих ребра: «ложное» и «истинное», а также выделенную временную переменную, которая управляет ветвлением. Эта управляющая переменная всегда должна иметь размер в 1 бит и определяться в текущем блоке или каком-либо его доминаторе. В случае, когда в реальности

из базового блока исходит только одно ребро (безусловный переход), в качестве управляющей переменной выбирается специальная переменная с номером 0, которая всегда имеет «ложное» значение. Если базовый блок должен был бы иметь более чем 2 перехода, такой блок заменяется каскадом ветвлений.

Все анализируемые в рамках какой-либо задачи базовые блоки объединяются в *таблицу базовых блоков*, где за ними закрепляются индексы. Таким образом, ссылки на последующие блоки хранятся в виде таких индексов.

5.2.6. Фрагменты

Наиболее крупной структурной единицей промежуточного представления является *фрагмент*. Логически фрагмент представляет собой направленный граф с единственным входом и единственным выходом, т.е. гамак. Вершинами в этом графе являются базовые блоки, а ребра соответствуют возможным передачам управления. Таким образом, фрагмент может рассматриваться как подпрограмма.

Входной базовый блок может иметь непустую входную группу. В этом случае соответствующие переменные должны быть определены до начала работы с фрагментом и являются его входными параметрами. Аналогично, значения переменных из выходной группы выходного базового блока представляют собой результат работы фрагмента. Благодаря этому одни фрагменты могут вызывать другие при помощи оператора **CALL** или использоваться как обработчики исключительных ситуаций в операторах **LOAD.R** и **STORE.R**.

Фрагменты, так же, как и другие составные элементы промежуточного представления, объединяются в *таблицу фрагментов* и получают индексы для перекрестных ссылок.

5.2.7. Контексты

Все перечисленные таблицы: адресных пространств, операций, базовых блоков и фрагментов, в совокупности образуют *контекст*. Последующий анализ всегда ведется в рамках какого-либо контекста.

6. Анализ кода на базе декларативной модели процессора

Общее решение, позволяющее задать декларативную модель процессора и являющееся основной для проведения широкого спектра разновидностей анализа бинарного кода, может быть получено в результате комбинации предложенных выше в подразделах 4.2 и 5.2 подходов и идеи абстрактной интерпретации [19].

Рассмотрим сначала блоки функциональности, которые должна предоставлять модель процессора.

I. Декодирование бинарного кода целевой процессорной архитектуры. В зависимости от задачи источником бинарного кода может выступать статический образ скомпилированной программы, снимок памяти или последовательность выполняемых команд при динамическом анализе. Результатом работы этого блока является структура, описывающая декодированную команду, в частности ее длину, мнемонику, набор операндов (в т.ч. режимы адресации и конкретные номера регистров, адреса, смещения и т.п.).

II. Трансляция одной декодированной команды или блока из нескольких команд в промежуточное представление, описанное в подразделе 5.2. Для того чтобы осуществить трансляцию, следует использовать внешние спецификации процессоров, которые с командами целевых машин сопоставляют Pivot-фрагменты. В результате может быть получен Pivot-код, описывающий поведение одной или нескольких машинных команд.

III. Оптимизация полученного Pivot-кода. Так как единицей трансляции изначально является одна машинная команда, при компоновке фрагментов неизбежны избыточные вычисления, в частности вычисление адресов операндов. Для того чтобы уменьшить объем промежуточного представления, который будет в дальнейшем анализироваться, необходимо провести следующие оптимизации в пределах фрагмента:
   i. провести свертку и продвижение констант;
   ii. исключить общие подвыражения;
   iii. исключить избыточные операторы *LOAD.L* и *STORE.L* (при этом операторы *LOAD.R* и *STORE.R*, логика которых может менять в разных сценариях анализа, вообще говоря, не подлежат исключению).

IV. Для некоторых сценариев анализа требуется также моделировать поведение машины «на стыке» команд, а именно логику выборки очередной команды, обработку исключений и т.п. Это необходимо для полносистемного анализа, а также для более точного построения графа потока управления в случаях, когда целевая архитектура имеет аппаратную поддержку циклов и т.п.

Таким образом, декларативная модель процессора состоит из следующих частей:

- декодер команд, представленный в виде, описанном выше в разделе 4.2;
- набор Pivot-фрагментов, которые задают операционную семантику отдельных машинных команд, а также таблица сопоставления этих фрагментов с результатами декодирования;
- фрагмент Pivot-кода, описывающего логику процессора по обработке исключений и выборке очередной команды, т.е. действий, которые выполняются между командами.

При наличии базы таких декларативных моделей процессоров появляется возможность единообразной работы с бинарным кодом различных целевых процессорных архитектур посредством предоставления возможности абстрактной интерпретации поверх промежуточного представления.

Зафиксируем некоторую решетку абстрактных состояний L и зададим следующие две передаточные функции:

- передаточная функция для базовых блоков TB отображает абстрактное состояние на входе в блок в состояние на выходе;
- передаточная функция для ребер TE отображает абстрактное состояние на входе в ребро в состояние на выходе.

Обратим внимание, что эти определения предполагают задачу, в которой распространение состояния ведется в прямом направлении. В задаче с обратным направлением прохода по ребрам графа потока управления будут использоваться симметричные определения. Отметим также, что вместо функции T_B может быть задана функция T_S, работающая над отдельным оператором промежуточного представления. Функций T_B в этом случае будет строиться как композиция T_S для составляющих базовый блок операторов.

В совокупности решетка, заданные передаточные функции и начальное состояние во входной вершине дают некоторую постановку задачи анализа потока данных (и соответствующую абстрактную интерпретацию) для Pivot-фрагмента. Алгоритм, который строит MFP-решение этой задачи, не зависит от постановки, и будет являться универсальным. Более того, при указанном способе задания передаточных функций алгоритм может быть реализован двумя способами: с хранением состояний как для базовых блоков (классический вариант, описанный, например, в [28]), так и для ребер.

Указанный алгоритм, по сути, описывает подход к статическому анализу в универсальном виде. В зависимости от выбора решетки и передаточных функций он может использоваться для вычисления возможных значений переменных, статического анализа помеченных данных, поиска дефектов и т.д. Кроме того, на базе того же самого алгоритма могут быть реализованы перечисленные выше оптимизационные преобразования, которые устраняют артефакты трансляции бинарного кода в промежуточное представление.

Символьное выполнение может быть реализовано похожим способом. Так как символьное выполнение предполагает анализ каждого пути в отдельности (и, в идеале, построение MOP-решения задачи анализа потока данных), но при этом каждый выполняемый оператор также рассматривается в рамках некоторой абстрактной интерпретации, поставленная в указанном виде задача анализа потока данных может использоваться и в этом случае. Основным отличием будет необходимость реализации диспетчера, который выбирает очередной просматриваемый путь при достижении ветвления в программе.

Порядок: статика, динамика, частный случай динамики – символьное выполнение, частный случай символьного выполнения – конкретное выполнение.

Наконец, динамический анализ может рассматриваться как упрощенный вид символьного выполнения, когда для каждого ветвления известна ветвь, по которой в дальнейшем пойдет управление. Таким образом, и динамический анализ является частным случаем абстрактной интерпретации и может быть

проведен на базе описанных выше конструкций. Отдельно отметим задачу конкретной интерпретации кода: нетрудно видеть, что путем отождествления конкретных и абстрактных состояний и выбором соответствующего вида решетки L можно проводить конкретную интерпретацию на базе абстрактной.

Наконец, добавим, что в силу того, что декартово произведение решеток является решеткой, в рамках одного запуска может проводиться анализ сразу нескольких аспектов поведения программы. Это, в частности, интересно при динамическом анализе (например, одновременно с конкретным выполнением программы может проводиться анализ доступа к неинициализированным ячейкам памяти, анализ примитивов синхронизации и т.п.) и при смешанном выполнении, когда часть программы или программной системы выполняется символьно, а часть – конкретно.

7. Заключение

В данной работе систематизированы задачи и подходы в анализе бинарного кода. С учетом особенностей современных процессорных архитектур и опыта существующих средств декодирования машинных команд и описания их операционной семантики, предложено новое промежуточное представление для анализа бинарного кода. Показано, что данное представление может быть использовано единообразно в различных задачах статического, динамического анализа бинарного кода и при символьном выполнении.

К дальнейшей работе авторы относят реализацию предложенных идей в виде набора библиотек с открытым исходным кодом и формирование базы декларативных описаний распространенных процессоров.

Список литературы

[1]. Wang X., Zeldovich N., Kaashoek M. F., Solar-Lezama A. A Differential Approach to Undefined Behavior Detection. ACM Transactions on Computer Systems, vol. 33, no. 1, art. 1, 2015, 29 p. DOI: 10.1145/2699678.

[2]. Nethercote N., Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM SIGPLAN Notices, 2007, vol. 42, no. 6, pp. 89-100.

[3]. Chipounov V., Candea G. Enabling sophisticated analyses of x86 binaries with RevGen. In Proc. of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W), 2011, pp. 211-216.

[4]. Lattner C., Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In Proc. of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, 2004, pp. 75-86.

[5]. Song D., Brumley D., Yin H., Caballero J., Jager I., Kang M.G., Liang Z., Newsome J., Poosankam P., Saxena P. BitBlaze: A new approach to computer security via binary analysis. Information systems security, 2008, pp. 1-25.

[6]. Падарян В.А., Соловьев М.А., Кононов А.И. Моделирование операционной семантики машинных инструкций. Программирование, том 37, № 3, 2011, стр. 50-64.

[7]. Brumley D., Jager I., Avgerinos T., Schwartz E.J. BAP: a binary analysis platform. Computer Aided Verification, 2011, pp. 463-469.
[8]. Dullien T., Porst S. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In Proc. of the CanSecWest Conference, 2009.
[9]. Bellard F. QEMU, a fast and portable dynamic translator. In Proc. of the USENIX Annual Technical Conference, 2005.
[10]. Luk C.K., Cohn R., Muth R., Patil H., Klauser A., Lowney G., Wallace S., Reddi V.J., Hazelwood K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. ACM SIGPLAN Notices, vol. 40, no. 6, 2005, pp. 190-200.
[11]. Bruening D., Amarasinghe S. Efficient, transparent, and comprehensive runtime code manipulation. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.
[12]. Chipounov V., Kuznetsov V. S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems. In Proc. of the 16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS. 2011.
[13]. Cha S. K., Avgerinos T., Rebert A., Brumley D. Unleashing mayhem on binary code. IEEE Symposium on Security and Privacy (SP), 2012, pp. 380-394.
[14]. Падарян В.А., Каушан В.В., Федотов А.Н. Автоматизированный метод построения эксплойтов для уязвимости переполнения буфера на стеке. Труды ИСП РАН, том 26, вып. 3, 2014, стр. 127-144. DOI: 10.15514/ISPRAS-2014-26(3)-7.
[15]. Kruegel C., Valeur F., Robertson W., Vigna G. Static Analysis of Obfuscated Binaries. In Proc. of the 13th USENIX Security Symposium, 2004, pp. 255-270.
[16]. Ben Khadra M. A., Stoffel D., Kunz W. Speculative disassembly of binary code. In Proc. of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, 2016.
[17]. Balakrishnan G., Reps T. Analyzing Memory Accesses in x86 Executables. In Proc. of the 13th International Conference on Compiler Construction, 2004, pp. 5-23.
[18]. Aslanyan H., Asryan S., Hakobyan J., Vardanyan V., Sargsyan S., Kurmangaleev S. Multiplatform Static Analysis Framework for Program Defects Detection. In Proc. of the CSIT Conference, 2017.
[19]. Cousot P., Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proc. of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1977, pp. 238-252.
[20]. Padaryan V.A., Getman A.I., Solovyev M.A., Bakulin M.G., Borzilov A.I., Kaushan V.V., Ledovskikh I.N., Markin Yu.V., Panasensko S.S. Methods and software tools to support combined binary code analysis. Programming and Computer Software, vol. 40, no. 5, 2014, pp. 276-287.
[21]. GNU Binutils. URL: http://sourceware.org/binutils/, дата обращения: 03.12.2018.
[22]. Capstone. URL: http://www.capstone-engine.org/, дата обращения: 03.12.2018.
[23]. IDA Pro. URL: https://www.hex-rays.com/products/ida/index.shtml, дата обращения: 03.12.2018.
[24]. Fauth A., Van Praet J., Freericks M. Describing instruction set processors using nML. European Design and Test Conference, 1995, pp. 503-507.
[25]. Hadjiyiannis G., Hanono S., Devadas S. ISDL: An instruction set description language for retargetability. In Proc. of the 34th annual Design Automation Conference, 1997, pp. 299-302.

[26]. Fox A. Improved tool support for machine-code decompilation in HOL4. In Proc. of the International Conference on Interactive Theorem Proving, 2015, pp. 187-202.
[27]. Gray K.E., Kerneis G., Mulligan D., Pulte C., Sarkar S., Sewell, P. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In Proc. of the 48th International Symposium on Microarchitecture, 2015, pp. 635-646.
[28]. Muchnick S.S. Advanced Compiler Design & Implementation. Morgan Kaufmann Publishers, 1997.

Next generation intermediate representations for binary code analysis

[1,2] *M.A. Solovev <icee@ispras.ru>*
[1] *M.G. Bakulin <bakulinm@ispras.ru>*
[1] *M.S. Gorbachev <sadbear@ispras.ru>*
[1,2] *D.V. Manushin <dman95@ispras.ru>*
[1,2] *V.A. Padaryan <vartan@ispras.ru>*
[1] *S.S. Panasenko <spanasenko@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*

Abstract. A lot of binary code analysis tools do not work directly with machine instructions, instead relying on an intermediate representation from the binary code. In this paper, we first analyze problems in binary code analysis that benefit from such an IR and construct a list of requirements that an IR suitable for solving these problems must meet. Generally speaking, a universal binary analysis platform requires two principal components. The first component is a retargetable instruction decoder that utilizes external specifications for describing target instruction sets. External specifications facilitate maintainability and allow for quickly adding support for new instruction sets. We analyze some of the more common ISAs, including those used in microcontrollers, and from that produce a list of requirements for a retargetable decoder. We then survey existing multi-ISA decoders and propose our vision of a more generic approach, based on a multi-layered directed acyclic graph describing the decoding process in universal terms. The second component of an analysis platform is the actual architecture-neutral IR. In this paper we describe such existing IRs, and propose Pivot 2, an IR that is low-level enough to be easily constructed from decoded machine instructions, and at the same time is also easy to analyze. The main features of Pivot 2 are explicit side effects, SSA variables, a simpler alternative to phi-functions, and an extensible elementary operation set at the core. The IR also supports machines that have multiple memory address spaces. Finally, we propose a way to tie the decoder and the IR together to fit them to most binary code analysis tasks through abstract interpretation on top of the IR. The proposed scheme takes into account various aspects of target architectures that are overlooked in many other works, including pipeline specifics (handling of delay slots, hardware loop support, etc.), exception and interrupt management,

and a generic address space model where accesses may have arbitrary side effects due to memory-mapped devices or other non-trivial behavior of the memory system.

Keywords: abstract interpretation; binary code analysis; compiler techniques; dynamic analysis; software reverse engineering; static analysis; symbolic execution.

DOI: 10.15514/ISPRAS-2018-30(6)-3

For citation: Solovev M.A., Bakulin M.G., Gorbachev M.S., Manushin D.V., Padaryan V.A., Panasenko S.S. Next generation intermediate representations for binary code analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 6, 2018, pp. 39-68 (in Russian). DOI: 10.15514/ISPRAS-2018-30(6)-3

References

[1]. Wang X., Zeldovich N., Kaashoek M. F., Solar-Lezama A. A Differential Approach to Undefined Behavior Detection. ACM Transactions on Computer Systems, vol. 33, no. 1, art. 1, 2015, 29 p. DOI: 10.1145/2699678.

[2]. Nethercote N., Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM SIGPLAN Notices, 2007, vol. 42, no. 6, pp. 89-100.

[3]. Chipounov V., Candea G. Enabling sophisticated analyses of x86 binaries with RevGen. In Proc. of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W), 2011, pp. 211-216.

[4]. Lattner C., Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In Proc. of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, 2004, pp. 75-86.

[5]. Song D., Brumley D., Yin H., Caballero J., Jager I., Kang M.G., Liang Z., Newsome J., Poosankam P., Saxena P. BitBlaze: A new approach to computer security via binary analysis. Information systems security, 2008, pp. 1-25.

[6]. Padaryan V.A., Solov'ev M.A., Kononov A.I. Simulation of operational semantics of machine instructions. Programming and Computer Software, vol. 37, no. 3, 2011, pp. 161-170. DOI: 10.1134/S0361768811030030.

[7]. Brumley D., Jager I., Avgerinos T., Schwartz E.J. BAP: a binary analysis platform. Computer Aided Verification, 2011, pp. 463-469.

[8]. Dullien T., Porst S. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In Proc. of the CanSecWest Conference, 2009.

[9]. Bellard F. QEMU, a fast and portable dynamic translator. In Proc. of the USENIX Annual Technical Conference, 2005.

[10]. Luk C.K., Cohn R., Muth R., Patil H., Klauser A., Lowney G., Wallace S., Reddi V.J., Hazelwood K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. ACM SIGPLAN Notices, vol. 40, no. 6, 2005, pp. 190-200.

[11]. Bruening D., Amarasinghe S. Efficient, transparent, and comprehensive runtime code manipulation. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[12]. Chipounov V., Kuznetsov V. S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems. In Proc. of the 16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS. 2011.

[13]. Cha S. K., Avgerinos T., Rebert A., Brumley D. Unleashing mayhem on binary code. IEEE Symposium on Security and Privacy (SP), 2012, pp. 380-394.

[14]. Padaryan V.A., Kaushan V.V., Fedotov A.N. Automated exploit generaton method for stack buffer overflow vulnerabilities. Trudy ISP RAN/Proc. ISP RAN, vol. 26, no. 3, 2014, pp. 127-144 (in Russian). DOI: 10.15514/ISPRAS-2014-26(3)-7.

[15]. Kruegel C., Valeur F., Robertson W., Vigna G. Static Analysis of Obfuscated Binaries. In Proc. of the 13th USENIX Security Symposium, 2004, pp. 255-270.

[16]. Ben Khadra M. A., Stoffel D., Kunz W. Speculative disassembly of binary code. In Proc. of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, 2016.

[17]. Balakrishnan G., Reps T. Analyzing Memory Accesses in x86 Executables. In Proc. of the 13th International Conference on Compiler Construction, 2004, pp. 5-23.

[18]. Aslanyan H., Asryan S., Hakobyan J., Vardanyan V., Sargsyan S., Kurmangaleev S. Multiplatform Static Analysis Framework for Program Defects Detection. In Proc. of the CSIT Conference, 2017.

[19]. Cousot P., Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proc. of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1977, pp. 238-252.

[20]. Padaryan V.A., Getman A.I., Solovyev M.A., Bakulin M.G., Borzilov A.I., Kaushan V.V., Ledovskikh I.N., Markin Yu.V., Panasensko S.S. Methods and software tools to support combined binary code analysis. Programming and Computer Software, vol. 40, no. 5, 2014, pp. 276-287.

[21]. GNU Binutils. URL: http://sourceware.org/binutils/, дата обращения: 03.12.2018.

[22]. Capstone. URL: http://www.capstone-engine.org/, дата обращения: 03.12.2018.

[23]. IDA Pro. URL: https://www.hex-rays.com/products/ida/index.shtml, дата обращения: 03.12.2018.

[24]. Fauth A., Van Praet J., Freericks M. Describing instruction set processors using nML. European Design and Test Conference, 1995, pp. 503-507.

[25]. Hadjiyiannis G., Hanono S., Devadas S. ISDL: An instruction set description language for retargetability. In Proc. of the 34th annual Design Automation Conference, 1997, pp. 299-302.

[26]. Fox A. Improved tool support for machine-code decompilation in HOL4. In Proc. of the International Conference on Interactive Theorem Proving, 2015, pp. 187-202.

[27]. Gray K.E., Kerneis G., Mulligan D., Pulte C., Sarkar S., Sewell, P. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In Proc. of the 48th International Symposium on Microarchitecture, 2015, pp. 635-646.

[28]. Muchnick S.S. Advanced Compiler Design & Implementation. Morgan Kaufmann Publishers, 1997.