DOI: 10.15514/ISPRAS-2019-31(6)-4

Декодирование машинных команд в задаче абстрактной интерпретации бинарного кода

[1,2] М.А. Соловьев, ORCID: 0000-0002-0530-6442 <icee@ispras.ru>
[1] М.Г. Бакулин, ORCID: 0000-0002-8569-7382 <bakulinm@ispras.ru>
[2] С.С. Макаров, ORCID: 0000-0003-0077-237X <smakarov@ispras.ru>
[2] Д.В. Манушин, ORCID: 0000-0001-8985-4114 <dman95@ispras.ru>
[1,2] В.А. Падарян, ORCID: 0000-0001-7962-9677 <vartan@ispras.ru>

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

Аннотация. Программный инструментарий для работы с бинарным кодом востребован не только разработчиками: невозможно добиться достаточной безопасности современных программ без изучения свойств исполняемого кода. Базовым компонентом такого инструментария является декодер машинных команд. У разных процессорных архитектур реализации декодеров разнородны, результаты разбора команд несовместимы, а сопровождение затруднительно из-за повсеместной практики реализовывать декодеры в виде каскадов ветвлений. Дальнейший анализ бинарного кода (на уровне потоков данных и управления, символьная интерпретация и др.) оказывается непереносимым между различными процессорными архитектурами из-за ограничений и особенностей реализации декодеров. В статье предлагается подход к декодированию машинных команд, основанный на внешних спецификациях. Отличительной особенностью подхода является оригинальный способ представления декодированной команды в универсальном (т.е. не отличающемся от одной архитектуры к другой) виде. Декодирование осуществляется при помощи разработанной авторами абстрактной стековой машины. Несмотря на неизбежные накладные расходы, обусловленные большей универсальностью подхода, прототип реализации показывает скорость разбора лишь в 1,5-2,5 раза уступающую традиционным декодерам, с учетом времени разбора спецификации и формирования служебных структур данных. Предлагаемый подход к организации декодирования позволит в перспективе развернуть единый стек программных средств анализа бинарного кода, применимый к различным процессорным архитектурам. В статье обсуждаются вопросы дальнейшей трансляции декодированных команд в машинно-независимое промежуточное представление для анализа их операционной семантики и проведения абстрактной интерпретации. Приведены практически полезные примеры интерпретации: конкретная интерпретация для эмуляции бинарного кода и «направляющая» интерпретация, позволяющая увязать идею абстрактной интерпретации с задачей углубленного автоматического анализа отдельных путей в программе.

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

Для цитирования: Соловьев М.А., Бакулин М.Г., Макаров С.С., Манушин Д.В., Падарян В.А. Декодирование машинных команд в задаче абстрактной интерпретации бинарного кода. Труды ИСП РАН, том 31, вып. 6, 2019 г., стр. 65–88. DOI: 10.15514/ISPRAS–2019–31(6)–4

Благодарности: Работа поддержана грантом РФФИ № 18-07-01256.

Decoding of machine instructions for abstract interpretation of binary code

[1,2] M.A. Solovev, ORCID: 0000-0002-0530-6442 <icee@ispras.ru>
[1] M.G. Bakulin, ORCID: 0000-0002-8569-7382 <bakulinm@ispras.ru>
[2] S.S. Makarov, ORCID: 0000-0003-0077-237X <smakarov@ispras.ru>
[2] D.V. Manushin, ORCID: 0000-0001-8985-4114 <dman95@ispras.ru>
[1,2] V.A. Padaryan, ORCID: 0000-0001-7962-9677 <vartan@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. Not only developers require tools that work with binary code: it is impossible to achieve sufficient security in contemporary software without inspecting its properties at this level. A key component of binary code analysis toolset is the instruction decoder. Different instruction set architectures give rise to decoders that are differently structured, the decoding results are incompatible, and maintenance is hindered because of the ubiquitous practice of implementing decoders as cascades of conditional operators. Further binary code analysis (control and data flows, symbolic interpretation, etc.) cannot easily be ported from one target architecture to another because of limitations and peculiarities of decoder implementations. In this paper, we propose an approach to decoding machine instructions that is based on external specifications. The main distinction is an original way of representing the decoder instruction universally, i.e. in a way that does not differ from one architecture to another. The decoding process is handled by an abstract stack machine we have developed. Despite the inevitable overhead stemming from the approach's universality, an implementation prototype displays only 1.5-2.5 times slowdown compared to traditional decoders; the measurements include time required to parse the specification and build the required data structures. The proposed approach to organizing decoding would allow, in the long run, to establish a unified stack of binary code analysis tools that would be applicable to different instruction set architectures. The paper further discusses questions of translating the decoded instructions into a machine-neutral internal representation for analyzing their operational semantics and carrying out abstract interpretation. We give examples of practically useful interpretations: the concrete interpretation and a "directing" interpretation that allows to tie the idea of abstract interpretation with the problem of deeper automatic analysis of individual paths in a program.

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

For citation: Solovev M.A., Bakulin M.G., Makarov S.S., Manushin D.V., Padaryan V.A. Decoding of machine instructions for abstract interpretation of binary code. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 6, 2019, pp. 65-88 (in Russian). DOI: 10.15514/ISPRAS-2019-31(6)-4

Acknowledgements: This work was supported by RFBR grant no. 18-07-01256.

1. Введение

В исторической ретроспективе задачи, связанные с бинарным кодом, решались в основном в контексте разработки низкоуровневого системного программного обеспечения (ПО), такого как ядра операционных систем, драйверы, компоненты встраиваемого ПО микроконтроллеров и т.п. С развитием языков программирования высокого уровня, улучшением качества компиляторов и наращиванием производительности вычислительных систем интерес к бинарному коду постепенно сокращался. Однако в последние десятилетие тема анализа и преобразования бинарного кода вновь стала крайне актуальной. Не в последнюю очередь это связано с увеличением степени внимания к безопасности ПО и качеству кода.

Одна из важнейших задач – оценка критичности дефектов в ПО – в большинстве случаев может быть решена только на уровне бинарного кода. Так, вопрос о том, приводит ли

неопределенное поведение в программе на языке Си к эксплуатируемой уязвимости нельзя решить, оставаясь в рамках модели абстрактной машины этого языка: например, оценка последствий переполнения буфера требует учитывать то, каким образом компилятор разместил этот буфер и другие переменные, какой бинарный код он сгенерировал для соответствующих обращений. Из-за этих факторов можно наблюдать взрывное появление систем анализа бинарного кода с целью обнаружения и оценки критичности уязвимостей, в т.ч. основанных на идеях символьного выполнения.

Кроме того, оценка безопасности ПО, поставляемого без исходных кодов, однозначно требует наличия развитого инструментария анализа бинарного кода. В силу того, что бинарный код более объемен и неудобен для понимания человеком, чем соответствующий исходный код на языке программирования высокого уровня, к этому инструментарию предъявляются повышенные требования с точки зрения структуризации бинарного кода и его представления в виде, пригодном для экспертной оценки человеком.

В соответствии с обзором типовых сценариев анализа бинарного кода (как связанных с обеспечением безопасности, так и иных сценариев), проведенным авторами ранее в [1], можно выделить три их группы, основываясь на требованиях к уровню представления анализируемого кода:

- сценарии, в которых требуется определять границы машинных команд в сегменте кода, проводить их декодирование и поверхностный анализ, например, итеративное дизассемблирование [2], некоторые разновидности двоичного динамического инструментирования [3], онлайн-анализ с применением аппаратной виртуализации [4];
- сценарии, в которых требуется отслеживать зависимости по данным и управлению, индуцируемые машинными командами, например, построение срезов (слайсинг) [5], анализ помеченных данных [6], интерактивное восстановление схемы работы алгоритма [7];
- сценарии, в которых требуется анализировать операционную семантику машинных команд и более крупных функциональных блоков, например, задачи поиска и оценки программных дефектов и активации условных переходов [8], а также многие оптимизационные преобразования, востребованные в классической и JIT-компиляции, а также в динамической двоичной трансляции [9].

Нетрудно заметить, что каждая следующая группа сценариев в этом списке предъявляет нарастающие требования к уровню представления анализируемого кода. Например, для того чтобы описать операционную семантику команды, необходимо предварительно ее декодировать. С другой стороны, описанная операционная семантика команды содержит информацию о возникающих при ее выполнении зависимостях по данным и управлению.

Таким образом, отдельный интерес представляет задача декодирования машинных команд, а именно определение границ команд и представление каждой из них в виде, пригодном для дальнейшего анализа. Такой вид, как минимум, должен включать код операции и описание операндов. Традиционный подход к декодированию машинных команд предполагает для каждой целевой процессорной архитектуры реализовывать отдельный программный модуль декодирования. Результатом работы такого модуля будет являться структура, описывающая декодированную команду, причем конкретный вид структуры будет зависеть от целевой процессорной архитектуры. Краткий обзор устройства декодеров, входящих в состав известных инструментов, работающих на уровне бинарного кода, дан во разд. 2 статьи.

В разд. 3 предлагается подход к декодированию, где не только кодировки команд, но и их структура (возможные коды операций, модификаторы, режимы операндов) задаются внешними спецификациями, что позволяет единообразно осуществлять декодирование машинных команд разнородных процессорных архитектур.

Для решения задач, относящихся ко второй и третьей группе сценариев, как правило требуется трансляция полученных команд в архитектурно-независимое промежуточное представление. То, каким образом осуществляется такая трансляция, описано в разд. 4.

В разд. 5 изложен разработанный способ и кратко описана спроектированная программная инфраструктура для проведения абстрактной интерпретации на базе архитектурно-независимого промежуточного представления.

Разд. 6 содержит заключение и перечисление направлений дальнейших работ.

2. Декодирование машинных команд в существующих инструментах

В подавляющем большинстве инструментов, которые работают с бинарным кодом, декодирование команд реализовано в виде отдельных программных модулей. Эти модули разбирают кодировки команд и представляют результат в виде структур, вид которых определяется целевой процессорной архитектурой.

Среди библиотек, которые предоставляют возможность декодирования команд различных процессорных архитектур, наиболее известными являются библиотека libopcodes, входящая в состав пакета binutils [10] (и соответствующий инструмент печати декодированных команд objdump), и декодер Capstone [11].

В binutils результатом декодирования команды является структура *disassemble_info*, содержащая признак успешного разбора, класс команды с точки зрения наличия передачи управления и доступа к памяти, и непрозрачную часть, которая не описывается в доступном пользователю программном интерфейсе и различается от одной целевой архитектуры к другой. После успешного декодирования команды структуру *disassemble_info* можно передать в одну из функций печати команд (они существуют по одной для каждой поддерживаемой архитектуры) для получения текстового ассемблерного представления. Отдельно получить информацию о мнемонике и операндах команды нельзя. Разные модули декодирования написаны на языке Си и организованы по-разному. Так, модуль поддержки RISC-V управляется таблицей, содержащей маски битовых полей, на основании которых определяется тип команды и ее операнды. Модуль поддержки x86 состоит из перемежающихся таблиц и отдельных функций (как правило, выглядящих как каскады ветвлений), осуществляющих рекурсивный разбор отдельных частей команды. В текущей версии binutils (2.33.1) он состоит из 16583 строк кода без учета подключаемых заголовков.

Capstone применяет подход с внешними таблицами, элементы которых описывают изменение состояния декодера. Вид этих таблиц отличается между модулями поддержки разных архитектур. Частично эти таблицы и код обхода позаимствованы из проекта LLVM [12], однако в них внесены изменения для нужд задачи декодирования (основной сценарий использования таблиц кодировок в LLVM – для кодогенерации в компиляторном тракте). Например, комплект таблиц для архитектуры в версии Capstone 4.0.1 состоит из 4007 строк сгенерированного Си-кода. Скрипты для генерации табличного кода не входят в состав Capstone, внесение изменений возможно либо вручную, либо через взаимодействие с автором библиотеки. Результат декодирования представляется в виде структуры *cs_insn*, которая содержит идентификатор мнемоники команды, ее размер в байтах, текстовый вид мнемоники и операндов (причем не каждого отдельно, а в виде единой строки). Более детальная информация доступна в виде дополнительных структур, по одной для каждой поддерживаемой процессорной архитектуры. Набор сведений в этих структурах отличается между архитектурами и позволяет, в частности, выяснить, какие модификаторы (префиксы, суффиксы и т.п.) присутствовали в команде, а также, как именно были закодированы ее операнды.

Некоторые инструменты, работающие с бинарным кодом, вовсе не поддерживают в явном виде структуру декодированной команды. Например, эмулятор QEMU [13] сразу в процессе декодирования команды строит промежуточное представление TCG. Опять-таки, разные модули реализованы по-разному. В модуле для RISC-V имеется таблица декодирования, которая при помощи скрипта на языке Python переводится в Си-код. Модуль для x86 реализован вручную в виде набора функций, каждая из которых, по сути, представляет собой каскад операторов *switch*.

Аналогично устроен процесс как декодирования, так и обратного кодирования машинного кода в системе динамического двоичного инструментирования Valgrind [14]. Другая система инструментирования, Pin [3], использует библиотеку Intel XED [15], которая осуществляет кодирование и декодирование x86-команд. В этой библиотеке используются внешние спецификации для описания кодировок команд, однако их вид специфичен для системы кодирования x86.

Разрабатываемая в ИСП РАН среда анализа бинарного кода ТРАЛ [16] имеет отдельные модули для декодирования команд различных архитектур, но при этом результат может быть представлен в виде единой структуры *Instruction*. Эта структура содержит признак успешного декодирования, размер декодированной команды в байтах, идентификатор мнемоники, набор флагов (является ли команда передачей управления, вызовом, возвратом и т.п.), количество слотов задержки, строковый вид команды (отдельно для префиксов, мнемоники и каждого из операндов). Кроме того, каждый операнд описывается структурой, в которой содержится: значение операнда (если это константа) или элемент какого-либо адресного пространства, в т.ч. регистрового файла (если это регистр или операнд в памяти). Модули декодирования реализованы вручную на языках Си/Си++ и в конечном счете также сводятся к разбору отдельных полей кодировки по таблицам, описанным явно или в виде каскада ветвлений.

Таким образом, практически все описанные системы не используют внешние спецификации для декодирования команд, либо используют их только в рамках отдельной процессорной архитектуры. В сочетании с тем, что результаты декодирования команд представляются в разнородном виде, для углубленного исследования (например, анализ помеченных данных, поиск ошибок, символьное выполнение, инструментирование) бинарного кода с использованием этих инструментов потребуется дополнительная прослойка, обеспечивающая еще один уровень трансляции в общий вид промежуточного представления. В рамках данной работы, в сущности, исследуется вопрос возможности обобщения вида такого рода спецификаций для того, чтобы исключить или сократить объем кодовой обвязки для такого рода трансляции.

В завершение обзора отметим, что данная работа не является первой в этом направлении. Например, в статье [17] решается похожая задача, однако для DSP-процессоров, которые, как правило, имеют существенно более регулярные и простые по структуре наборы команд. В разработанной в АНБ и выложенной в открытый доступ в апреле 2019 г. системе Ghidra [18] применяются внешние спецификации, описывающие как процесс декодирования, так и последующее построение промежуточного представления. Декодирование управляется правилами, которые описывают предикат над полями кодировки. Если предикат истинный, то применяется соответствующее правило. Правило описывает текстовый вид команды и ее семантику в виде последовательности операторов промежуточного представления.

Отметим, что на момент начала работы над методами декодирования и трансляции в промежуточное представление, излагаемыми в данной статье, система Ghidra еще не была доступна для ознакомления (в частности, наша работа [19] была опубликована за несколько месяцев до выпуска Ghidra).

3. Декодирование машинных команд по внешним спецификациям

Для того чтобы обеспечить декодирование машинных команд по внешним спецификациям, необходимо спроектировать два компонента:

- модель архитектуры набора команд (ISA), описывающую структуру команд и задающую возможные кодировки для выбранной целевой машины;
- программный компонент, осуществляющий декодирование отдельных команд по такому описанию.

При таком подходе поддержка новой архитектуры набора команд будет сводиться к подготовке ее спецификации в соответствии с разработанной моделью (т.е. пополнению некоторой базы знаний), а изменения в программный код декодера вносить не потребуется.

3.1 Модель архитектуры набора команд

При описании архитектуры набора команд необходимо учесть следующие ключевые особенности, сформулированные ранее в работе [1]:

- могут существовать несовместимые версии архитектуры набора команд (например, в MIPS), а также расширения, влияющие на разбор определенных кодировок (например, расширение “C” в RISC-V);
- кодировки команд могут иметь переменную длину, которая не известна до начала декодирования и определяется самой кодировкой команды (например, в x86 – от 1 до 15 байтов, в RISC-V с расширением “C” – 2 или 4 байта, в наборе команд T32 архитектуры ARM);
- одна и та же последовательность битов может быть декодирована по-разному в зависимости от значений управляющих битов (например, признак разрядности кода в x86, активный набор команд в ARM).

Сформулируем первую из перечисленных особенностей в более общем виде: в архитектуре набора команд присутствуют параметры, влияющие на то, как конкретная реализация этой архитектуры (т.е. конкретный процессор) воспринимает определенные кодировки. Для того чтобы поддержать эту особенность необходимо в явном виде указать эти параметры для каждой архитектуры набора команд. Таким образом, первой составляющей модели будет являться перечень таких параметров. Каждый параметр назовем *характеристикой* (*feature*) и зафиксируем его идентификатор и длину в битах. При задании значения характеристики для конкретного процессора потребуется указать битовый вектор соответствующей длины. Например, характеристиками в рамках такого определения являются:

- поддержка семейства команд AVX в x86 – логическое значение (1 бит), также определяет возможность кодирования команд при помощи VEX-префиксов;
- поддержка расширения “M” в RISC-V – логическое значение (1 бит), определяет, доступны ли в данной реализации архитектуры RISC-V команды умножения, деления и взятия остатка;
- номер версии архитектуры MIPS (release) – целое число (3 бита), влияющее на наличие или отсутствие отдельных групп кодировок команд.

Теперь перейдем к вопросу о представлении результата декодирования, что позволит сформулировать еще две составляющие модели. Напомним, что основным отличием и целью предлагаемого подхода является единообразный процесс декодирования и, как следствие, единообразное представление его результатов (т.е. декодированных команд). В качестве такого представления предлагается структура, описывающая команду как набор *морфем* и последовательность операндов.

Под морфемой здесь понимается свойство, дающее вклад в описание поведения команды. Наиболее характерным примером морфемы является мнемоника, соответствующая коду операции. Однако во многих процессорных архитектурах мнемоника – не единственное такое свойство команды. Например, в наборе команд A32 архитектуры ARM большая часть команд может иметь поле “cond”, определяющее условие над флагами регистра состояния, при котором команда должна быть исполнена. Если это условие не выполняется, то команда отбрасывается. В ассемблерном виде это поле соответствует двухбуквенному суффиксу после мнемоники команды, например команда сложения “addal” выполняется всегда (при этом суффикс “al” может быть опущен – “add”), а команда “addcs” – только если флаг переноса содержит единицу.

Другим примером является суффикс «точка» в PowerPC – команды “add” и “add.” отличаются тем, будут ли при выполнении сложения обновляться флаги регистра состояния. Морфемами являются также суффиксы “.aq” и “.rl”, указывающие возможный порядок операций доступа к памяти в командах расширения “A” архитектуры RISC-V. Наконец, морфемой является префикс “lock”, меняющий свойства некоторых команд доступа к памяти в x86. Таким образом, набор морфем описывает операцию, которую выполняет команда, с учетом модификаторов. Понятно, что перечень морфем для каждой целевой архитектуры свой и является второй составляющей модели.

Помимо набора морфем для задания структуры декодированной команды необходим также формализм для операндов. Возможные режимы адресации существенно отличаются от одной архитектуры набора команд к другой. Например, в архитектуре RISC-V в качестве операндов могут выступать:

- регистр общего назначения (РОН) x0…x31;
- регистр для вычислений с плавающей точкой f0…f31;
- системный CSR-регистр, заданный 12-разрядным номером;
- непосредственно закодированная константа длиной 5, 6, 12 либо 20 битов;
- операнд в памяти, адресуемый при помощи одного из регистров общего назначения и 12-разрядного знакового смещения.

Видно, что с каждым из этих режимов операндов можно сопоставить набор *атрибутов*, конкретные значения которых и будут задавать операнд соответствующего режима. Так, для регистров общего назначения и регистров для вычислений с плавающей точкой единственным атрибутом будет 5-разрядный номер регистра, для CSR-регистра – 12-разрядный номер, для каждой разновидности констант – значение соответствующей разрядности. Наконец, операнду в памяти соответствуют два атрибута: номер базового регистра и значение смещения. Перечислив все возможные режимы операндов и задав для каждого из них перечень атрибутов, мы получим третью составляющую модели, а именно «выразительные средства» для описания операндов декодированных команд. В рамках зафиксированных для каждой целевой архитектуры режимов операндов декодированный операнд описывается: во-первых, идентификатором режима; во-вторых, кортежем битовых векторов, задающих конкретные значения атрибутов, соответствующих данному режиму.

Приведем несколько примеров представления декодированных команд для различных архитектур (для записи использована JSON-нотация):

- [["lwu"], ["reg", { "rid": 4 }], ["mem", { "rid": 13, "offset": 10 }]]:
  - архитектура RISC-V, ассемблерный вид: “lwu x4, 10(x13)”;
  - множество морфем состоит из единственного элемента;
  - имеется два операнда: с режимом “reg” (регистр общего назначения) и с режимом “mem” (операнд в памяти);
- [["ori"], ["reg", { "rid": 6 }], ["reg", { "rid": 8 }], ["imm16", { "value": 127 }]]:

  - архитектура MIPS, ассемблерный вид: “ori $6, $8, 127”;
  - множество морфем состоит из единственного элемента;
  - имеется три операнда: первый и второй с режимом “reg” (регистр общего назначения), третий с режимом “imm16” (16-разрядная непосредственно закодированная константа);
- [["subfze", "o", "."], ["reg", { "rid": 4 }], ["reg", { "rid": 2 }]]:
  - архитектура PowerPC, ассемблерный вид “subfzeo. r4, r2”;
  - множество морфем состоит из мнемоники “subfze” и модификаторов “o” и “.”;
  - имеется два операнда с режимом “reg” (регистр общего назначения);
- [["lock", "xadd"], ["mem16a32", { "sri": 3, "bri": 2, "bri_p": 1, "iri": 0, "iri_p": 0, "scale": 0, "disp": 0, "disp_sz": 0 }], ["reg16", { "rid": 6 }]]:
  - архитектура x86, ассемблерный вид “lock xadd word [edx], si”;
  - множество морфем состоит из мнемоники “xadd” и префикса “lock”;
  - имеется два операнда: операнд в памяти с режимом “mem16a32” (размер операнда 16 битов, размер адреса 32 бита) и регистр общего назначения с режимом “reg16”.

Как можно видеть, предложенный способ описания декодированной команды подходит для большого набора разнородных архитектур набора команд. Теперь мы имеем возможность более конкретно поставить задачу декодирования команд: декодер должен принимать на вход поток битов (соответствующий одной или нескольким командам, заданным полностью или частично) и текущие значения управляющих битов, влияющих на декодирование, а на выходе формировать либо структуру команды как набор морфем и последовательность операндов, а также возвращать длину кодировки в битах, либо выдавать признак ошибки, если команда не может быть декодирована.

Эту задачу можно решить отдельно для каждой целевой архитектуры набора команд (что будет соответствовать традиционному подходу к декодированию) или задать правила, по которым формируются кодировки команд, в рамках модели архитектуры. Предлагаемое решение соответствует выбору второго пути; основная мотивация такого выбора связана с тем, что поддержка и пополнение базы знаний (спецификаций целевых процессорных архитектур) представляется менее затратным процессом, чем доработка и отладка существующих и реализация новых программных компонентов.

3.2 Декодирование машинных команд

Для описания правил, по которым в рамках заданной архитектуры формируются кодировки команд, разработан формализм, основанный на абстрактной стековой машине. Состояние стековой машины характеризуется:

- содержимым стека, каждый элемент которого является битовым вектором, векторы могут иметь различную длину;
- текущим набором выданных морфем;
- текущей последовательностью выданных операндов.

Состояние стековой машины меняется в ответ на применение *правил*, которые задаются в рамках модели архитектуры набора команд. Каждое правило относится к одному из двух типов: командное или вспомогательное правило. Совокупность всех командных правил описывает допустимые кодировки команд. Вне зависимости от своего типа, правило характеризуется количеством входных и выходных аргументов, каждый из которых – битовый вектор. Предполагается, что на момент применения правила на стеке абстрактной машины расположены значения входных аргументов правила (глубже по стеку могут располагаться произвольные дополнительные элементы), а при его успешном

применении на момент выхода входные аргументы будут сняты со стека, а вместо них размещены значения выходных аргументов правила.

С точки зрения выполняемых абстрактной машиной шагов каждое правило является последовательностью *случаев*, а каждый случай, в свою очередь, последовательностью *действий*. Перечислим возможные действия и то, как их выполнение влияет на состояние абстрактной машины, а далее перейдем к случаям и целым правилам.

- Действие *APPLY-BIT-VEC-OPERATION* применяет одну из простых операций над битовыми векторами (из словаря BTOR[1], определенного в работе [20]) к операндам, расположенным на стеке. Действие выполняется успешно, если операция может быть применена к такому сочетанию операндов. В этом случае операнды снимаются со стека, а вместо них на стек помещается результат применения операции. В противном случае абстрактная машина переходит в особое состояние *REJECT*, соответствующее невозможности продолжения работы.
- Действие *APPLY-RULE* применяет указанное вспомогательное правило. Как было указано выше, входные аргументы правила находятся на стеке; если правило может быть успешно применено, то после его отработки входные аргументы будут сняты со стека, а вместо них положены выходные. В противном случае абстрактная машина переходит в состояние *REJECT*.
- Действие *CHECK-FEATURE* вычисляет логическое выражение над конкретными значениями характеристик (features) процессора. Если результат вычисления – ложь, то абстрактная машина переходит в состояние *REJECT*. Иными словами, последующие действия могут выполняться только в том случае, если описанная комбинация характеристик либо истинна, либо может оказаться истинной (в случае, когда значения каких-либо характеристик неизвестны).
- Действие *DUP* копирует элемент с указанном индексом в стеке и добавляет эту копию в качестве новой вершины стека. Если стек короче, чем указанный индекс, то абстрактная машина переходит в состояние *REJECT*.
- Действие *EMIT-MORPHEME* добавляет в набор выданных морфем указанную морфему. Данное действие всегда выполняется успешно.
- Действие *EMIT-OPERAND* добавляет в набор выданных операндов операнд с указанным режимом, значения атрибутов которого последовательно снимаются со стека. Если длины или количество атрибутов на стеке не совпадают с заданными в модели для данного режима, абстрактная машина переходит в состояние *REJECT*.
- Действие *GATHER* снимает со стека один битовый вектор и по указанной маске собирает его отдельные биты в виде нового битового вектора, который кладется на стек. Если стек пуст или вектор на его вершине имеет недостаточную длину для применения указанной маски, абстрактная машина переходит в состояние *REJECT*.
- Действие *MATCH-PATTERN* проверяет соответствие битового вектора на вершине стека с указанным шаблоном. Шаблон представляет собой последовательность битов, где каждый бит имеет либо конкретное значение (0 или 1), либо допускает любое значение бита в данной позиции. Если стек пуст либо битовый вектор на вершине стека не соответствует шаблону, абстрактная машина переходит в состояние *REJECT*.

[1] BTOR представляет собой способ описания SMT-формул на основе словаря базовых арифметических и логических операций. Операции были отобраны для описания поведения дискретной аппаратуры в рамках задачи проверки моделей и работают над битовыми векторами различной длины. В частности, SMT-решатель Boolector использует BTOR как основное представление формул.

- Действие *POP* снимает со стека указанное количество элементов. Если на стеке меньше указанного числа элементов, абстрактная машина переходит в состояние *REJECT*.
- Действие *PUSH* кладет на вершину стека указанный константный битовый вектор. Данное действие всегда выполняется успешно.
- Действие *SCATTER* снимает со стека один битовый вектор и по указанной маске распределяет его отдельные биты в виде нового битового вектора, который кладется на стек. Если стек пуст или вектор на его вершине имеет недостаточную длину для применения указанной маски, абстрактная машина переходит в состояние *REJECT*.

Как было сказано выше, каждый случай представляет собой последовательность действий. Абстрактная машина меняет свое состояние при выполнении случая, последовательно выполняя эти действия. Если после выполнения очередного действия абстрактная машина переходит в состояние *REJECT*, то и выполнению всего случая будет соответствовать состояние *REJECT*. Иными словами, случай выполняется успешно тогда и только когда, когда все действия в нем последовательно выполняются успешно.

Наконец, при выполнении правила последовательно рассматривается каждый его случай. Абстрактная машина пытается выполнить очередной случай, и если это успешно удается (т.е. результатом не является состояние *REJECT*), то правило в целом считается успешно выполненным и полученное состояние становится состоянием после выполнения правила в целом. Иначе восстанавливается состояние на момент начала выполнения правила и происходит рассмотрение следующего случая. Если ни один случай не выполнился успешно, то и все правило в целом считается не выполнившимся успешно, и на выходе из выполнения правила абстрактная машина переходит в состояние *REJECT*. Иными словами, правило выполняется успешно тогда и только тогда, когда существует хотя бы один случай в нем, который выполняется успешно, причем тогда выполнение правила эквивалентно выполнению первого из таких случаев.

Следует обратить внимание, что упорядоченность случаев в правиле важна для более компактного задания правил, и является заимствованием из PEG-грамматик [21]. Например, зачастую встречаются кодировки команд, где какое-либо поле имеет несколько значений, интерпретируемых особым образом, и остальные значения, которые соответствуют общему случаю. Размещая частные случаи перед общими, можно решить проблему пересечения описываемых значений без дополнительных проверок.

Таким образом, наивная реализация декодера по описанным правилам предполагает перебор всех возможных случаев. В подразделе 3.4 мы покажем, каким образом можно ускорить работу декодера за счет подготовки вспомогательных структур данных.

Все командные правила в рамках одной спецификации должны иметь один и тот же набор входных и выходных аргументов. Первым входным аргументом всегда является кодировка, разбор которой производится. Последующие аргументы (которых может и не быть) соответствуют отдельным управляющим битам, которые могут влиять на декодирование. Единственным выходным аргументом является длина в битах разобранной кодировки. Все командные правила могут быть объединены в одно общее правило путем конкатенации наборов случаев. Это общее командное правило и будет выполняться при декодировании очередной команды.

Таким образом, начальное состояние абстрактной машины при декодировании очередной команды следующее:

- на вершине стека расположен битовый вектор с кодировкой одной или нескольких команд, первую из которых необходимо декодировать (поскольку в общем случае заранее известна только максимальная длина команды для заданной архитектуры, этот вектор зачастую будет иметь именно такую длину);

- глубже по стеку расположены значения отдельных управляющих битов (полей системных регистров), которые могут влиять на декодирование;
- текущий набор выданных морфем пуст;
- текущая последовательность выданных операндов пуста.

Если при выполнении общего командного правила абстрактная машина не переходит в состояние *REJECT*, то:

- на вершине стека расположен битовый вектор, значение которого – битовая длина кодировки декодированной команды;
- текущий набор выданных морфем соответствует полному набору морфем команды;
- текущая последовательность выданных операторов соответствует полной последовательности операторов команды.

Рассмотрим два примера командных правил, соответствующих кодировкам команды “andi” архитектуры RISC-V.

- Команда “andi” кодируется вектором вида “*iiiiiiiiiii*_*mmmmm*_111_*ddddd*_0010011”, где поле *ddddd* задает номер целевого РОН, поле *mmmmm* – номер исходного РОН, а поле *iiiiiiiiiii* – значение операнда-константы. Соответствующее командное правило будет принимать на входе единственный вектор *input* (входную кодировку) и выдавать на выходе длину при успешном разборе команды. Правило состоит из единственного случая со следующей последовательностью действий:
  - *MATCH-PATTERN*(“*xxxxxxxxxxx*_*xxxxx*_111_*xxxxx*_0010011”) – проверка того, что фиксированные биты кодировки соответствуют требуемым – стек имеет вид [*input*];
  - *EMIT-MORPHEME*(“andi”) – выдача морфемы – стек имеет вид [*input*];
  - *DUP*(0) – копирование входной кодировки – стек имеет вид [*input*, *input*];
  - *GATHER*(“000000000000_00000_000_11111_0000000”) – выделение поля *ddddd* – стек имеет вид [*input*, *ddddd*];
  - *EMIT-OPERAND*(“reg”) – выдача первого регистрового операнда, номер регистра снимается с вершины стека, т.е. берется из поля *ddddd* – стек имеет вид [*input*];
  - *DUP*(0) – копирование входной кодировки – стек имеет вид [*input*, *input*];
  - *GATHER*(“000000000000_11111_000_00000_0000000”) – выделение поля *mmmmm* – стек имеет вид [*input*, *mmmmm*];
  - *EMIT-OPERAND*(“reg”) – выдача второго регистрового операнда, номер регистра снимается с вершины стека, т.е. берется из поля *mmmmm* – стек имеет вид [*input*];
  - *GATHER*(“111111111111_00000_000_00000_0000000”) – выделение поля *iiiiiiiiiii* – стек имеет вид [*iiiiiiiiiii*];
  - *EMIT-OPERAND*(“imm12”) – выдача последнего операнда-константы, значение снимается с вершины стека, т.е. берется из поля *iiiiiiiiiii* – стек становится пустым;
  - *PUSH*(32) – на стек заносится длина кодировки успешно декодированной команды – стек имеет вид [32].
- При поддержке процессором расширения “C” команда “andi” может также быть закодирована в 16-разрядном виде “100_*i*_10_*rrr*_*iiiii*_01”. Эта кодировка требует, чтобы в качестве целевого и исходного регистра фигурировал один и тот же регистр, причем его номер должен быть в диапазоне от 8 до 15 включительно, а операнд-константа не может превышать 6 битов с учетом последующего знакового расширения. Правило для этой кодировки также будет состоять из единственного случая со следующей последовательностью действий (после тире указано состояние стека при успешном выполнении действия):

  - *CHECK-FEATURE*(“C”) – [*input*];
  - *MATCH-PATTERN*(“100_*x*_10_*xxx*_*xxxxx*_01”) – [*input*];
  - *EMIT-MORPHEME*(“andi”) – [*input*];
  - *DUP*(0) – [*input*, *input*];
  - *GATHER*(“000_0_00_111_00000_00”) – [*input*, *rrr*];
  - *APPLY-BIT-VEC-OPERATOR*(extu.2) – [*input*, 00*rrr*];
  - *PUSH*(8) – [*input*, 00*rrr*, 8];
  - *APPLY-BIT-VEC-OPERATOR*(or) – [*input*, 01*rrr*];
  - *DUP*(0) – [*input*, 01*rrr*, 01*rrr*];
  - *EMIT-OPERAND*(“reg”) – [*input*, 01*rrr*];
  - *EMIT-OPERAND*(“reg”) – [*input*];
  - *GATHER*(“000_1_00_000_11111_00”) – [*i_iiiii*];
  - *APPLY-BIT-VEC-OPERATOR*(exts.6) – [(*iiiiii*)_*i_iiiii*];
  - *EMIT-OPERAND*(“imm12”) – [];
  - *PUSH*(16) – [16].

3.3 Язык внешних спецификаций

Предложенный вид задания правил является достаточно удобным для реализации абстрактной машины декодирования, но явно неудобен для описания человеком. В связи с этим был разработан более язык более высоко уровня для спецификации архитектур набора команд, а также транслятор, который преобразует спецификации на этом языке в описанные в предыдущем подразделы сущности (правила, случаи, действия и т.п.).

Не приводя полную грамматику и правила интерпретации языковых конструкций, покажем общий вид языка на рис. 1. Здесь с его помощью описаны рассмотренные выше кодировки команды “andi” архитектуры RISC-V вместе с определениями морфем, режимов операндов и характеристик.

Ключевые слова *rule* и *insn* определяют, соответственно, вспомогательные и командные правила. Перед любым из них может быть указано ключевое слово *match*, которое меняет синтаксис задания правила на табличный. В приведенном примере в табличном синтаксисе заданы оба командных правила, а вспомогательное правило “RVC_REG” задано обычным способом как набор случаев и (высокоуровневых) действий. Конкретно в этом правиле случай всего один, поэтому в фигурных скобках сразу перечисляются действия. Помимо измененного синтаксиса, в табличные правила автоматически дописывается действие, возвращающее на стеке длину считанной кодировки.

```
feature C: '1;

morpheme andi;

opmode reg { rid: '5 }
opmode imm12 { imm: '12 }

match insn RV32I(input) -> (isz) [
    |iiiiiii_iiiii_mmmmm_111_ddddd_0010011| => {
        andi;
        reg { rid: d };
        reg { rid: m };
        imm12 { imm: i };
    },
]

match insn RVC64_Q1(input) -> (isz) feature C [
    |100_i_10rrr_iiiii_01| => {
        andi;
        RVC_REG(r);
        RVC_REG(r);
        imm12 { imm: 'exts.6(i) };
    },
]

rule RVC_REG(crid) -> () {
    reg { rid: 'or('exts.2(crid), 8'5)) };
}
```

Рис. 1. Фрагмент спецификации архитектуры набора команд RISC-V
Fig. 1. RISC-V ISA specification fragment

3.4 Реализация декодера

В рамках данной работы был реализован программный компонент, осуществляющий декодирование команд по внешним спецификациям. Компонент, в сущности, реализует описанную выше абстрактную стековую машину, однако имеет некоторые особенности, которые позволяют ему работать несколько быстрее:

- перед тем, как осуществлять декодирование команд, требуется задать значения для всех характеристик, т.е. перейти от архитектуры набора команд вообще к частной ее реализации (процессору);
- в рамках этого перехода статически вычисляются все значения предикатов в действиях *CHECK-FEATURE*; действия, где предикат истинный, удаляются, а действия, где предикат ложный, удаляются вместе с объемлющим случаем;
- таким образом, в полученной модификации модели описаны только те кодировки, которые соответствуют заданному набору значений характеристик, а действия *CHECK-FEATURE* не встречаются;
- если все случаи какого-либо правила начинаются с действия *MATCH-PATTERN* (характерная ситуация для табличного сравнения), то для этого правила создается специальная структура, ускоряющая выбор нужного случая.

Структура для ускорения логически представляет собой таблицу шаблонов, с каждым из которых сопоставлено число – номер случая. Структура отвечает на вопрос «какому номеру случая соответствует данный входной битовый вектор?», причем в качестве положительного ответа возвращается номер случая, а в качестве отрицательного ответа –

признак отсутствия подходящего случая. Физически эта таблица представлена в виде дерева со следующими типами вершин:

- *ACCEPT* – принимает любой вход и возвращает указанный номер случая;
- *REJECT* – не принимает никакой вход и возвращает отрицательный ответ;
- *FORK* – по очереди применяет вход к своим потомкам до тех пор пока не получит положительный ответ или потомки не закончатся (что соответствует отрицательному ответу);
- *DENSE* – выделяет по маске битовое поле из входа, интерпретирует его как номер потомка и применяет вход к нему;
- *SPARSE* – выделяет по маске битовое поле из входа, интерпретирует его как ключ в хеш-таблице, значения которой ссылаются на потомков, и применяет вход к соответствующему потомку, либо возвращает отрицательный ответ, если хеш-таблица не содержит искомый ключ.

Структура дерева строится при помощи вспомогательного компонента, который принимает на вход пары вида (битовый шаблон, номер случая), а также выбранную стратегию построения.

Линейная стратегия построения соответствует поведению абстрактной стековой машины декодирования без оптимизаций: вершина дерева имеет тип *FORK*, а каждому случаю соответствует вершина типа *ACCEPT* с соответствующим номером.

Жадная стратегия построения работает следующим образом. Сначала для каждой битовой позиции подсчитывается, в скольких шаблонах этот бит определен (т.е. его значение влияет на выбор соответствующего случая). На основании этих подсчетов формируется очередь битовых последовательностей в порядке невозрастания указанной метрики. Далее начинает применяться следующая рекурсивная процедура.

- Если множество шаблонов пусто, то формируется дерево из единственной вершины типа *REJECT*.
- Если множество шаблонов состоит из единственного тривиального шаблона (т.е. такого, в котором любой бит может иметь любое значение), то формируется дерево из единственной вершины типа *ACCEPT*, а номер соответствующего случая берется из единственного тривиального шаблона.
- В остальных случаях из очереди извлекается следующая рассматриваемая битовая позиция. Множество шаблонов разбивается на три подмножества: P_0 – те шаблоны, где на рассматриваемой битовой позиции нулевой бит, P_1 – те шаблоны, где на рассматриваемой битовой позиции единичный бит, и P_x – те шаблоны, где на рассматриваемой битовой позиции произвольный бит. При этом разбиении во множествах P_0 и P_1 бит на рассматриваемой позиции заменяется на произвольный. Процедура вызывается рекурсивно для каждого из трех множеств с оставшейся частью очереди, в результате чего формируются три поддерева T_0, T_1 и T_x. Они, в свою очередь, объединяются в общее дерево по следующим правилам.
  - Сначала объединяются поддеревья T_0 и T_1. Если оба корня – *REJECT*, то и результат объединения тоже *REJECT*.
  - Если один из корней – *DENSE*, а второй – *DENSE* с такой же маской или *REJECT* и можно объединить случаи в общую *DENSE*-вершину, у которой будет не более, чем некоторое пороговое число потомков (на настоящий момент – 256), то это объединение происходит. При этом логически (но не физически) *REJECT*-корень (если он есть) заменяется на *DENSE*-поддерево с соответствующим количеством *REJECT*-потомков. Далее два *DENSE*-поддерева объединяются путем выбора потомков то одного, то второго из них в правильном порядке в зависимости от того, где находится текущая рассматриваемая битовая позиция относительно маски

потомков.

- В остальных случаях формируется *DENSE*-поддерево с поддеревьями T_0 и T_1, а маска формируется из текущей битовой позиции.
- После того, как поддеревья T_0 и T_1 объединены в поддерево T_{01}, оно объединяется с T_x. Если одно из этих поддеревьев имеет *REJECT*-корень, то результатом объединения становится второе. В противном случае создается *FORK*-поддерево с потомками T_{01} и T_x (в данном порядке).

Наконец, *жадная разреженная* стратегия отличается от жадной тем, что при построении дерева *DENSE*-вершины, в которых более чем половина поддеревьев является *REJECT*-поддеревьями, заменяются на эквивалентные *SPARSE*-вершины (т.е. массив с «пустыми» элементами заменяется на хеш-таблицу). Эта стратегия имеет такую же амортизированную вычислительную сложность, что и жадная стратегия, однако позволяет несколько сократить объем используемой памяти.

Тестирование производительности реализованного декодера было проведено для двух наборов команд: 64-разрядная модель RISC-V и 64-разрядная модель PowerPC. В качестве входных файлов выступали «сырые» (без заголовков) секции кода исполняемых файлов и библиотек из образов Debian Linux. Для RISC-V использовался образ от 18.04.2018 г. [22], а для PowerPC – от 16.11.2019 г. [23]. Чтобы обеспечить объективность тестирования, вокруг разработанного декодера была реализована обвязка, которая выводит декодированные команды в текстовом виде в соответствии с ассемблерным синтаксисом, который принят для каждой из двух рассматриваемых архитектур. В тестировании также участвовали две сборки утилиты objdump из набора binutils [10] версии 2.33 (для RISC-V и для PowerPC), а также декодер Capstone [11] версии 4.0.1 для PowerPC и декодер RV8 [24] версии от 23.09.2018 г. для RISC-V. Каждый тестируемый инструмент запускался на каждом входном файле 10 раз подряд, после чего высчитывалось среднее время запуска. Тестирование проводилось на компьютере с процессором AMD Ryzen 5 1400 3.20ГГц и 32ГиБ оперативной памяти под управлением 64-разрядной ОС Ubuntu Linux 19.10. Результаты тестирования приведены на рис. 2 (RISC-V) и рис. 3 (PowerPC). Горизонтальная ось графиков – среднее время одного запуска в миллисекундах (логарифмическая шкала).

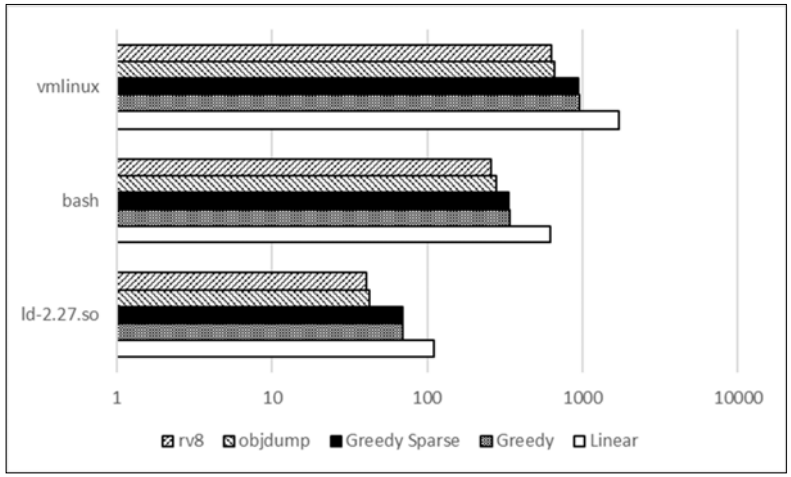

Рис. 2. Сравнение производительности декодирования на наборе команд RISC-V
Fig. 2. Decode benchmark for RISC-V ISA

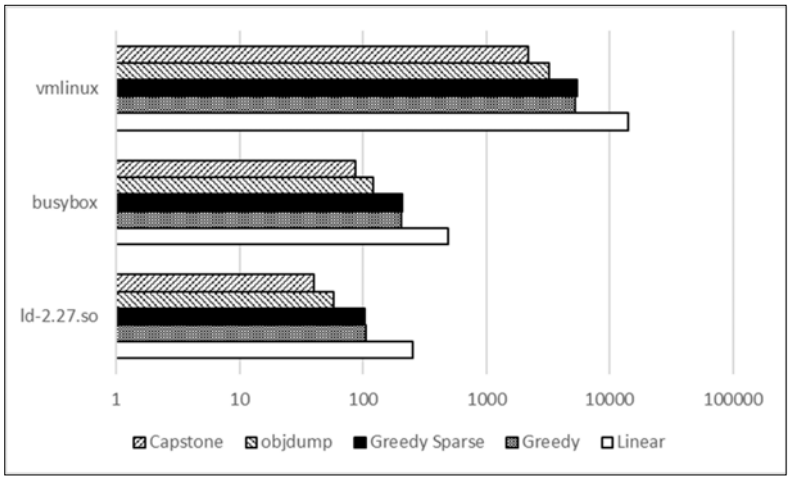

Рис. 3. Сравнение производительности декодирования на наборе команд PowerPC
Fig. 3. Decode benchmark for PowerPC ISA

Видно, что во всех случаях скорость разработанного декодера уступает реализованным вручную и оптимизированным под конкретные наборы команд традиционным декодерам. В особенности это касается случаев, когда использовалась линейная стратегия. Однако при применении жадной или жадной разреженной стратегии отставание по скорости от objdump в среднем находится в диапазоне 1,5-2,5 раза. Следует учесть, что для разработанного декодера тестировался вариант с «полным циклом»: чтение внешней спецификации на исходном языке, ее трансляция в правила для абстрактной стековой машины и дальнейшее декодирование команд по этим правилам.

Проведенное профилирование показывает, что в реализации есть пространство для дальнейшей оптимизации за счет уменьшения количества копирований и динамического выделения памяти, что дает надежду на достижение времен, близких к показываемым реализованными вручную традиционными декодерами. Кроме того, заметный вклад во время работы разработанного декодера вносит также отдельный этап формирования строкового представления команды, в то время как в традиционных декодерах этот этап совмещен с самим декодированием.

4. Трансляция в промежуточное представление

Углубленный анализ поведения бинарного кода предполагает его трансляцию в машинно-независимое промежуточное представление. В предыдущей работе [1] нами составлены требования к такому представлению и предложено реализующее их представление Pivot 2. Кратко напомним, что Pivot 2 является представлением в форме статического единичного присваивания, однако вместо φ-функции применяется подстановка значений переменных на ребрах. Все переменные представляют собой битовые векторы различной длины. Наиболее крупная единица представления – модуль – состоит из фрагментов, соответствующих функциональным блокам с единственной входной и единственной выходной вершиной.

Фрагменты могут иметь входные и выходные параметры-битовые векторы. С точки зрения внутренней структуры фрагмент представляет собой компоненту связности графа потока управления. Каждый базовый блок в нем состоит из последовательности операторов: *CALL* (вызов фрагмента), *INIT* (инициализация переменной константным

битовым вектором), *INVOKE* (применение операции над переменными-битовыми векторами), *LOAD.L* (загрузка из локального адресного пространства), *LOAD.R* (загрузка из удаленного адресного пространства), *MIX* (переименование переменных), *SLICE* (конкатенация и выделение полей битовых векторов[2]), *STORE.L* (выгрузка в локальное адресное пространство) и *STORE.R* (выгрузка в удаленное адресное пространство). Адресными пространствами в терминах Pivot 2 являются любые адресуемые области памяти, включая регистровые файлы, порты ввода-вывода и т.д. Под локальным адресным пространством понимается такое, которое может быть корректно промоделировано как буфер памяти (например, регистры общего назначения в большинстве процессоров), причем любой доступ к такому пространству завершается успешно. Удаленное адресное пространство имеет произвольную семантику доступов, некоторые из которых могут завершаться с ошибкой.

Трансляция отдельно взятой машинной команды в промежуточное представление сводится к следующему. Необходимо описать фрагмент, который моделирует операционную семантику этой команды. При этом входными параметрами фрагмента становятся значения атрибутов декодированной и представленной в описанном выше виде команды. Выбор требуемого фрагмента осуществляется на основе аннотаций, в которых для фрагментов, описывающих поведение машинных команд, указывается:

- набор морфем: все перечисленные морфемы должны входить в множество, которое сформировал декодер в результате разбора;
- режимы адресации операндов: для каждого операнда режим адресации должен совпасть с тем, который получен в результате разбора.

В качестве примера рассмотрим уже упоминавшуюся выше команду “andi” процессорной архитектуры RISC-V. Эта команда осуществляет операцию «побитовое И», входными операндами которой являются 32-, 64- или 128-разрядный РОН (в зависимости от варианта архитектуры) и 12-разрядная непосредственно закодированная константа, которая подлежит знаковому расширению до размера РОН. Результат помещается в выходной РОН (номер которого может совпадать или не совпадать с входным). На рис. 4 приведен представленный на высокоуровневом языке[3] фрагмент для этой команды для случая 64-разрядных РОН.

Аннотация *isa::insn* перед фрагментом указывает форму команды, к которой применим этот фрагмент: множество морфем должно включать морфему “addi”, команда должна иметь три операнда, первые два из которых относятся к режиму “reg”, а третий – “imm12”. При аргументах функции присутствуют аннотации *isa::subst*, указывающие, значением какого атрибута декодированной команды должен быть инициализирован соответствующий аргумент.

В процессорной архитектуре RISC-V РОН с номером 0 имеет особенный смысл: записи в него не происходят, а чтения всегда возвращают 0. Описанный фрагмент учитывает это поведение: если номер выходного регистра *rd* – нулевой, то команда игнорируется (1). В противном случае во временной переменной *a* формируется значение первого входного операнда: если это нулевой РОН, то нулевое значение, а иначе значение, считанное из адресного пространства регистров по смещению, соответствующему номеру регистра (2). Переменная *b* получает значение второго входного операнда как знаковое расширение *imm* (3). Наконец, результат операции «побитовое И» записывается в адресное пространство регистров (4).

[2] На момент публикации работы [1] для этих действий использовались отдельные операторы *CONCAT* и *EXTRACT*, которые позже были объединены в оператор *SLICE*.

[3] Синтаксис языка не является окончательным; на момент написания статьи над ним ведется активная работа.


```
#[isa::insn "andi(reg, reg, imm12)"]
fn andi(
    #[isa::subst "0.rid"]
    rd: '5,
    #[isa::subst "1.rid"]
    rs1: '5,
    #[isa::subst "2.imm"]
    imm: '12,
) {
    if zero(rd) {
        // (1)
    } else little {
        let a = if zero(rs1) { 0'64 }
                    else { regs[mul(rs1, 8'5)] };  // (2)
        let b = exts(imm);  // (3)
        regs[mul(rd, 8'5)] = and(a, b);  // (4)
    }
```

Рис. 4. Спецификация операционной семантики команды “addi” набора команд RISC-V
Fig. 4. Operational semantics specification for the “addi” instruction of the RISC-V ISA

Поскольку в разные моменты доступ даже к одним и тем же адресным пространствам может предполагать разный порядок байтов, любое обращение должно быть аннотировано конкретным порядком. В данном случае это происходит при помощи объемлющего блока *little*, предписывающего порядок little endian.

Имея фрагмент, описывающий семантику команды, транслировать ее в промежуточное представление легко: достаточно вызвать этот фрагмент с нужными параметрами. Однако такой подход не очень эффективен: видно, что для конкретного экземпляра команды “addi” большая часть вычислений может быть сделана статически (во время трансляции): могут быть удалены ветвления, константа может быть заранее расширена, а адреса в пространстве регистров заранее вычислены. В связи с этими соображениями более эффективно осуществить специализацию фрагмента путем подстановки его содержимого в объемлющий фрагмент вместе с инициализацией значений входных переменных, а затем провести оптимизационные преобразования:

- удаление общих подвыражений, в т.ч. избыточных доступов к локальным адресным пространствам;
- свертку и продвижение констант;
- удаление мертвого кода и пустых базовых блоков.

Следует отметить, что особенно заметен выигрыш от оптимизационных преобразований будет при трансляции блоков машинных команд в виде одного фрагмента: в этом случае общие подвыражения будут удаляться в рамках всего транслируемого блока. Трансляция в этом случае сводится к последовательной подстановке фрагментов, соответствующих отдельным командам, и оптимизации результирующего фрагмента. При этом, конечно, транслируемый блок должен иметь единственный вход. Если у блока несколько выходов, то потребуется возвращать из фрагмента номер выхода для дальнейшей диспетчеризации.

5. Абстрактная интерпретация промежуточного представления

Напомним, что в классическом определении [25] абстрактная интерпретация состоит из:

- полной полурешетки абстрактных состояний *A*-*Cont* с операцией ◦ и индуцированным ей отношением частичного порядка ≤, нижним и верхним элементами;

- отображения *Int*, переводящего набор абстрактных состояний на входных ребрах базового блока в абстрактные состояния на выходных ребрах.

В классическом определении предполагается, что отображение *Int* сохраняет порядок (или, в терминах задач потока данных, является монотонной передаточной функцией). Тогда по теореме Кнастера-Тарского [26] при итеративном применении отображения *Int* ко всем точкам программы соответствующее «глобальное» отображение для состояния всей программы имеет неподвижную точку. При этом предполагается, что решается задача оценки поведения программы в целом, т.е. задача статического анализа.

В рамках разрабатываемой инфраструктуры анализа бинарного кода предлагается в части случаев использовать более слабое понимание абстрактной интерпретации, разбив его на два.

Под *интерпретацией* в контексте данной работы будем понимать:

- состояние – произвольный тип данных, описывающий некоторые аспекты состояния анализируемой программы, и поддерживающий создание копий;
- набор передаточных функций, отображающих входное состояние либо в выходное, либо в особое «невозможное» состояние:
  - функция *edge*, соответствующая проходу по ребру;
  - функция *call_entry*, соответствующая подготовительным действиям перед вызовом фрагмента (формированию значений входных аргументов перед началом интерпретации вызванного фрагмента);
  - функция *call_exit*, соответствующая завершающим действиям после вызова фрагмента (формированию значений выходных аргументов по окончании интерпретации вызванного фрагмента);
  - функции *init*, *invoke*, *load_local*, *load_remote*, *mix*, *slice*, *store_local*, *store_remote*, соответствующие выполнению операторов *INIT*, *INVOKE*, *LOAD.L*, *LOAD.R*, *MIX*, *SLICE*, *STORE.L*, *STORE.R*.

Интерпретация также характеризуется *направлением* – прямым или обратным. Это влияет, в частности, на то, в каком порядке будут рассматриваться операторы в базовом блоке. Отметим, что декартово произведение любого набора интерпретаций также будет являться интерпретацией, причем ее состояние будет соответствовать декартову произведению состояний отдельных интерпретаций. В этом случае определим, что общее состояние будет считаться «невозможным» если хотя бы одно из состояний отдельных интерпретаций выродилось в «невозможное».

Под *монотонной интерпретацией* будем понимать интерпретацию, для которой выполнено дополнительно следующее:

- ее состояние является полной решеткой;
- все передаточные функции являются монотонными относительно частичного порядка, определяемого данной решеткой.

Можно видеть, что определение монотонной интерпретации совпадает с классическим определением абстрактной интерпретации (поскольку утверждения, что множество является полной решеткой и полной полурешёткой, эквивалентны). В монотонных интерпретациях «невозможное» состояние по определению соответствует нижнему элементу решетки состояний. Декартово произведение монотонных интерпретаций является монотонной интерпретацией.

5.1 Конкретная интерпретация

Построим в описанных терминах *конкретную интерпретацию*, т.е. такую интерпретацию, которая соответствует поведению бинарного кода при его реальном

запуске на некоторой машине. В сущности, решаемая задача является задачей эмуляции бинарного кода, однако в рассматриваемом случае она решается в рамках терминологии абстрактной интерпретации.

Состояние конкретной интерпретации состоит из двух аспектов: стека, в свою очередь состоящего из отдельных кадров, и состояний каждого адресного пространства. Каждый кадр стека содержит отображение из номера локальной переменной в ее значение. Значение переменной – это либо конкретный битовый вектор, либо признак «неизвестное значение» определенной битовой длины. Состояние адресного пространства – это непрозрачный объект, который изменяется в ответ на загрузку и выгрузку.

- Метод *load* данного объекта соответствует загрузке из пространства данных с указанным размером и порядком байтов, размещенных по указанному адресу. В случае успеха возвращается полученное значение переменной, в случае ошибки – дескриптор ошибки, также в виде значения переменной. В обоих случаях значение может быть неопределенным, аналогично значениям переменных в кадрах стека.
- Метод *store* соответствует выгрузке в пространство данных с указанным значением (определенным или нет) и порядком байтов по указанному адресу. В случае ошибки возвращается дескриптор ошибки.

Если метод *load* или *store* возвращает ошибку при доступе к локальному пространству, это интерпретируется как «невозможное» состояние. Поскольку все локальные пространства должны демонстрировать единообразное поведение, существует одна реализация такого объекта для поддержания состояния локального пространства. В основе реализации лежит разреженный массив байтов, выделяемый по необходимости отдельными страницами. Каждый бит содержит либо 0, либо 1, либо неопределенное значение.

Если для какого-либо пространства на момент начала интерпретации не задан объект состояния, то записи в это пространство игнорируются, а чтения всегда возвращают неопределенные значения.

Сама конкретная интерпретация сводится к реализации описанных выше передаточных функций и является, очевидно, прямой интерпретацией. Следует отметить, что функция *invoke* реализуется путем вычисления SMT-выражений, описывающих значения выходных аргументов в виде формул над значениями входных аргументов операций.

Конкретная интерпретация не является монотонной: переменная в рамках какого-либо кадра, равно как и участок адресного пространства, могут произвольным образом менять свои значения произвольное число раз (в случае переменной для этого необходимо, чтобы место ее статического единичного присваивания находилось внутри цикла). Таким образом, для ее применения требуется некоторый компонент, который будет отслеживать текущее состояние интерпретации и положение в интерпретируемом коде. Этот компонент описан в следующем подразделе.

5.2 Конкретный исполнитель

Конкретный исполнитель осуществляет применение выбранной прямой интерпретации вдоль одного пути в анализируемом коде. В начале работы конкретному исполнителю должны быть переданы:

- Pivot-модуль, в рамках которого будет проводиться интерпретация;
- начальная точка в модуле;
- выбранная прямая интерпретация;
- состояние интерпретации, соответствующее начальной точке.

После инициализации конкретный исполнитель ведет себя как *итератор*: он может продвигаться к следующему состоянию до тех пор, пока это состояние существует (т.е. не является «невозможным»). Каждый шаг происходит следующим образом.

- Если текущая точка соответствует какому-либо оператору, то происходит вызов соответствующей передаточной функции над текущим состоянием и полученное состояние становится текущим. Если рассматриваемый оператор – *CALL*, то его выполнение распадается на три части – подготовку вызова, интерпретацию вызванного фрагмента и завершение вызова.
- Если текущая точка соответствует концу базового блока, то для каждого ребра, исходящего из этого блока, вызывается передаточная функция *edge*. Если оба состояния «невозможные», то состояние исполнителя становится «невозможным». Если «невозможное» только одно из этих состояний, то состоянием исполнителя объявляется второе. Наконец, если оба состояния оказались возможными, то выбранная интерпретация не может быть продолжена в рамках конкретного исполнителя, поскольку появился второй возможный путь. В этом случае состояние исполнителя также становится «невозможным».

Конкретный исполнитель может применяться не только с конкретной интерпретацией. Он может также использоваться для проведения произвольного анализа в рамках некоторого фиксированного пути. В этом случае требуется таким образом задать интерпретацию, чтобы только вдоль интересующего пути она имела «возможные» состояния.

Например, если проводится динамический анализ по трассе и пройденный в рамках данного выполнения путь известен, направить произвольную интерпретацию вдоль этого пути можно следующим образом. Зададим еще одну прямую интерпретацию, состояние которой представляет собой единичный тип (т.е. тип с единственным значением). Будем считать, что это единственное значение соответствует «возможному» состоянию (напомним, что «невозможное» состояние существует отдельно от выбранного типа для состояния). Все передаточные функции задаваемой интерпретации, кроме функции *edge*, будут возвращать входное состояние (они будут вызываться только на «возможном» входном состоянии и, таким образом, возвращать «возможное» выходное состояние). Функция *edge*, рассматривая трассу, должна вернуть «возможное» состояние для пройденного исходящего ребра, и «невозможное» состояние для не пройденного исходящего ребра. Объединив построенную таким образом интерпретацию с произвольной заданной прямой интерпретацией через декартово произведение, мы получаем возможность проведения прямого динамического анализа. Симметричным образом можно организовать и обратный динамический анализ вдоль заданного пути. Отметим, что построенная «направляющая» интерпретация является монотонной.

Наконец, отметим, что если имеется какая-либо монотонная интерпретация, и она при помощи конкретного исполнителя применяется в связке с «направляющей» интерпретацией вдоль каждого возможного пути, то применяя операцию «сбор» к полученным в конечной точке состояниям, мы (по определению) построим MOP-решение соответствующей задачи потока данных.

6. Заключение

В настоящий момент все изложенные в данной статье методы и подходы в той или иной степени реализованы в экспериментальной инфраструктуре для абстрактной интерпретации бинарного кода Glassfrog, разрабатываемой в ИСП РАН. Дальнейшие работы связаны с улучшением существующих компонентов и разработкой и реализацией новых.

В части улучшения существующих компонентов планируется провести окончательную стабилизацию программного интерфейса и улучшить производительность компонента

декодирования машинных команд, выполнить его интеграцию с компонентом абстрактной интерпретации.

Помимо существующего конкретного исполнителя, планируется реализовать исполнитель для проведения статического анализа, позволяющий вычислить наибольшую или наименьшую неподвижную точку для заданной монотонной интерпретации. Также планируется реализация исполнителя для символьного выполнения, поддерживающего абстрактные состояния на множестве активных путей с внешней диспетчеризацией.

Список литературы / References

[1]. 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. Programming and Computer Software, vol. 45, issue 7, 2019, pp. 424-437. DOI: 10.1134/S0361768819070107.

[2]. 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, Article No. 16.

[3]. 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.

[4]. Ren S, Tan L, Li C, Xiao Z, Song W. Samsara: Efficient deterministic replay in multiprocessor environments with hardware virtualization extensions. In Proc. of the USENIX Annual Technical Conference, 2016, pp. 551-564.

[5]. Weiser M. Program slicing. IEEE Transactions on software engineering, vol. 10, issue 4, 1984, pp. 352-357.

[6]. Bakulin M, Klimushenkova M, Egorov D. Dynamic Diluted Taint Analysis for Evaluating Detected Policy Violations. Ivannikov ISPRAS Open Conference, 2017, pp. 22-26. DOI: 10.1109/ISPRAS.2017.00011.

[7]. Bugerya A.B., Kulagin I.I., Padaryan V.A., Solovev M.A., Tikhonov A.Yu. Recovery of High-Level Intermediate Representations of Algorithms from Binary Code. Ivannikov Memorial Workshop (IVMEM), 2019, pp. 57-63. DOI: 10.1109/IVMEM.2019.00015.

[8]. Федотов А.Н., Падарян В.А., Каушан В.В., Курмангалеев Ш.Ф., Вишняков А.В., Нурмухаметов А.Р. Оценка критичности программных дефектов в рамках работы современных защитных механизмов. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 73-92 / Fedotov A.N., Padaryan V.A., Kaushan V.V., Kurmangaleev Sh.F., Vishnyakov A.V., Nurmukhametov A.R. Software defect severity estimation in presence of modern defense mechanisms. Trudy ISP RAN/Proc. ISP RAN, vol. 28, issue 5, 2016, pp. 73-92 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-4.

[9]. Muchnick S.S. Advanced Compiler Design & Implementation. Morgan Kaufmann Publishers, 1997, 856 p.

[10]. GNU binutils. URL: https://www.gnu.org/software/binutils/, accessed 25.11.2019.

[11]. Capstone. URL: http://www.capstone-engine.org/, accessed 25.11.2019.

[12]. 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.

[13]. Bellard F. QEMU, a fast and portable dynamic translator. In Proc. of the USENIX Annual Technical Conference, 2005, pp. 41-46.

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

[15]. Intel XED. URL: https://intelxed.github.io/, accessed 25.11.2019.

[16]. 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. DOI: 10.1134/S0361768814050077.

[17]. Рубанов В.В., Михеев А.С. Интегрированная среда описания системы команд для сигнальных процессоров. Труды ИСП РАН, том 9, 2006, стр. 143-158 / Rubanov V.V, Mikheev A.S. Integrated environment for describing instruction systems of signal processors. Trudy ISP RAN/Proc. ISP RAN, vol. 8, 2006, pp. 143-158 (in Russian).

[18]. Ghidra. URL: https://www.nsa.gov/resources/everyone/ghidra/, accessed 25.11.2019.
[19]. 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, стр. 39-68 / 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 RAN, vol. 30, issue 6, 2018, pp. 39-68. DOI: 10.15514/ISPRAS-2018-30(6)-3.
[20]. Brummayer R, Biere A, Lonsing F. BTOR: bit-precise modelling of word-level problems for model checking. In Proc. of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning, 2008, pp. 33-38.
[21]. Ford B. Parsing expression grammars: a recognition-based syntactic foundation. In Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2004, pp. 111-122.
[22]. Debian Linux RISC-V OS image. URL: https://people.debian.org/~mafm/debian-riscv64-tarball-20180418.tar.xz, accessed 25.11.2019.
[23]. Debian Linux PowerPC OS image. URL: https://cdimage.debian.org/debian-cd/current/ppc64el/iso-cd/debian-10.2.0-ppc64el-netinst.iso, accessed 25.11.2019.
[24]. RV8. URL: https://rv8.io/, accessed 25.11.2019.
[25]. 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.
[26]. Tarski A. A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics, vol. 5, issue 2, 1955, pp. 285-309.

Информация об авторах / Information about authors

Михаил Александрович СОЛОВЬЕВ – кандидат физико-математических наук, старший научный сотрудник отдела компиляторных технологий ИСП РАН; старший преподаватель кафедры системного программирования факультета ВМК МГУ. Его научные интересы включают анализ бинарного и исходного кода, обратную инженерию ПО, операционные системы.

Mikhail Aleksandrovich SOLOVEV is a candidate of physical and mathematical sciences, senior researcher at the compiler technologies department of ISP RAS; senior lecturer at the system programming department of the faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University. His research interests include binary and source code analysis, software reverse engineering, and operating systems.

Максим Геннадьевич БАКУЛИН – младший научный сотрудник отдела компиляторных технологий ИСП РАН. Его научные интересы включают анализ бинарного и исходного кода, динамический анализ помеченных данных, символьное выполнение, эмуляцию и виртуализацию.

Maksim Gennadevich BAKULIN is a junior researcher at the compiler technologies department of ISP RAS. His research interests include binary and source code analysis, dynamic taint analysis, symbolic execution, emulation and virtualization.

Сергей Сергеевич МАКАРОВ – студент кафедры системного программирования ВМК МГУ. Его научные интересы включают анализ бинарного кода, эмуляцию и виртуализацию, операционные системы, обратную инженерию ПО.

Sergei Sergeevich MAKAROV is a student at the system programming department of the faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University. His research interests include binary code analysis, emulation and virtualization, operating systems and software reverse engineering.

Дмитрий Валерьевич МАНУШИН –аспирант кафедры системного программирования факультета ВМК МГУ. Его научные интересы включают анализ бинарного кода, анализ исходного кода, безопасность ПО.

Dmitrii Valerevich MANUSHIN is a postgraduate at the system programming department of the faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University. His research interests include binary code analysis, source code analysis and software security.

Вартан Андроникович ПАДАРЯН – кандидат физико-математических наук, ведущий научный сотрудник отдела технологий ИСП РАН; доцент кафедры системного программирования факультета ВМК МГУ. Его научные интересы включают компиляторные технологии, безопасность ПО, анализ бинарного кода, параллельное программирование, эмуляция и виртуализация.

Vartan Andronikovich PADARYAN is a candidate of physical and mathematical sciences, leading researcher at the compiler technologies department of ISP RAS; associate professor of the system programming department of the faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University. His research interests include compiler technologies, software security, binary code analysis, parallel programming, emulation and virtualization.