Санкт-Петербургский государственный университет

Кафедра системного программирования

Леденева Екатерина Юрьевна

Статический анализ бинарных модулей z/OS

Курсовая работа

Научный руководитель:
д. ф.-м. н., профессор Терехов А. Н.

Санкт-Петербург
2019

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering

Ekaterina Ledeneva

Static analysis of z/OS binaries

Course Work

Scientific supervisor:
professor Andrey Terekhov

Saint-Petersburg
2019

Оглавление

Введение

Статический анализ кода — метод получения разнообразной информации о программе без её исполнения.

Статический анализ используется для поиска уязвимостей в коде, верификации программного обеспечения, анализа поведения программ при отсутствии исчерпывающей документации, обратного инжиниринга. Анализ может производиться как над исходным кодом, так и над бинарными файлами. Потребность анализировать бинарный код возникает, например, при отсутствии доступа к исходному коду.

Большинство алгоритмов статического анализа программ определено над графом потока управления — графом, вершины которого содержат блоки инструкций, а рёбра представляют возможные потоки управления. Этот граф используется для выполнения анализа, а также помогает в понимании структуры исходного бинарного модуля. Некоторые приложения, использующие статические анализаторы, предоставляют пользователю более высокоуровневое, чем бинарный код, представление модуля.

Существуют различные продукты, в том числе с открытым исходным кодом, реализующие статический анализ, каждый из которых имеет как достоинства, так и недостатки. Поэтому необходимо проанализировать известные продукты, выбрать наиболее подходящий и расширить его функциональность для наших нужд. Так, большинство инструментов поддерживают только архитектуру процессора x86, однако позволяют встраивать в себя реализацию и иных архитектур, что и планируется сделать в рамках данной работы.

Анализ бинарных модулей вообще и модулей z/Architecture в частности имеет ряд проблем:

- Большое количество инструкций в архитектуре и сложность их семантики;
- Динамические переходы: до запуска программы в общем случае неизвестно, куда будут осуществляться переходы, так как их адре-

са могут храниться в регистрах, вычисляться по ходу исполнения и таким образом указывать на любые точки как внутри, так и за пределами анализируемого модуля. Это существенно затрудняет построение адекватного графа потока управления;

- Необходимость учитывать влияние инструкций не только на регистры общего назначения и память, но и на специальные регистры. Так, в z/OS некоторые команды меняют поля PSW[1], в частности, Condition Code, от содержимого которого могут зависеть дальнейшие переходы;

- Отсутствие разделения на сегменты кода и данных в z/OS.

Перечисленные проблемы делают задачу построения графа потока управления очень сложной, поэтому она сужается до построения приближения графа.

[1]Program status word, слово состояния программы.

1. Постановка задачи

Цель данной работы — реализация статического анализа бинарных модулей z/OS. Для её достижения были выделены следующие задачи:

- Исследование предметной области, сравнение существующих решений;
- Выбор одного из существующих решений в качестве основы для реализации;
- Разработка, реализация и встраивание в выбранную основу модуля, позволяющего обрабатывать бинарные файлы z/OS;
- Резализация графического интерфейса.

2. Обзор

В данной главе производится обзор двух из известных решений поставленной проблемы, их сравнение и, как результат, выбор одного их них как основы для реализации статического анализа бинарных модулей z/OS.

2.1. BAP

BAP (Binary Analysis Platform) — платформа бинарного анализа с открытым кодом, разработанная в университете Карнеги-Меллон [2, 1]. На данный момент BAP поддерживает архитектуры процессоров ARM, x86 и другие. Платформа написана на OCaml, предоставляется интерфейс для C, Python и Rust.

BAP — это фреймворк: для расширения его функциональности не надо менять его код, достаточно реализовать свой плагин. Помимо самого фреймворка предоставляется набор уже готовых инструментов, плагинов и библиотек. Некоторые из них:

- Beagle — деобфускатор строк — ищет строки, закодированные в бинарных файлах;
- Primus — интерпретатор промежуточного представления;
- Saluki — фреймворк верификации и обнаружения паттернов уязвимости;
- Taint Analysis — определяет, какие объекты могут быть изменены пользовательским вводом.

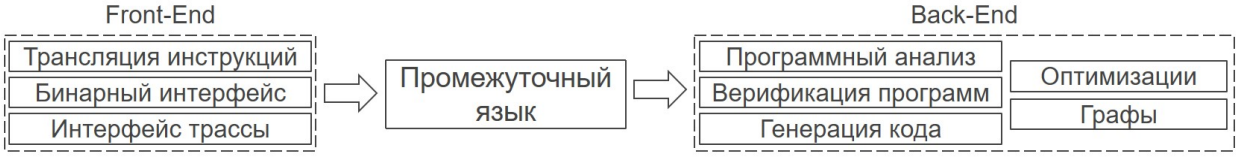


Рис. 1: Архитектура бинарного анализа и компоненты BAP [1]

BAP состоит из двух частей (рис. 1): front-end и back-end:

- Front-end отвечает за перевод бинарного кода в программу на промежуточном языке BIL[2]. Для этого используется дизассемблирование алгоритмом линейной развёртки (linear sweep). Семантика BIL полностью отражает семантику инструкций ассемблера.

- Back-end занимается непосредственно анализом полученного кода на промежуточном языке — строит граф потока управления, производит оптимизации и верификацию. Таким образом, back-end не зависит от архитектуры, и для встраивания некоторой архитектуры в BAP, необходимо реализовать для неё лишь первую часть — преобразование бинарного модуля в BIL.

BAP позволяет выводить результат анализа в форме графа потока управления, ассемблерного кода, набора инструкций BIL и других форматах.

Недостаток BAP заключается в том, что при построении графа потока управления он игнорирует косвенные переходы, что делает его анализ далёким от полноты. Для улучшения анализа предлагается использовать плагин, позволяющий использовать функции другого инструмента анализа программ — IDA Pro, который использует алгоритм рекурсивного обхода для дизассемблирования [7]. Однако это не является удовлетворительным решением проблемы по следующим причинам.

Во-первых, IDA Pro не поддерживает z/Architecture, так что требуется сначала реализовать встраивание в неё.

Во-вторых, IDA Pro — проприетарный продукт, в то время как нашей целью является расширение инструмента с открытым кодом. Это также накладывает ограничения на возможность модифицировать IDA для поддержки иных архитектур.

В-третьих, IDA Pro не умеет разрешать косвенные переходы в общем случае: она пользуется эвристиками для обнаружения в коде некоторых известных шаблонов, например, конструкции switch (путём поиска таблиц переходов).

[2]BAP Instruction Language

2.2. Jakstab

Jakstab (Java Toolkit for Static Analysis of Binaries) — дизассемблер и платформа для статического анализа бинарных модулей [8].

Jakstab реализован на Java и представляет собой фреймворк для построения графа потока управления. Jakstab имеет свой промежуточный язык в стиле RTL[3], в который он переводит дизассемблированные инструкции, используя библиотеку семантик. Дизассемблер Jakstab'а можно конфигурировать, таким образом добавляя поддержку желаемых архитектур [9].

Jakstab можно использовать как библиотеку или вызывать, используя интерфейс командной строки. В результате генерируются:

- Ассемблерный код;
- Граф потока управления;
- Автомат потока управления[4].

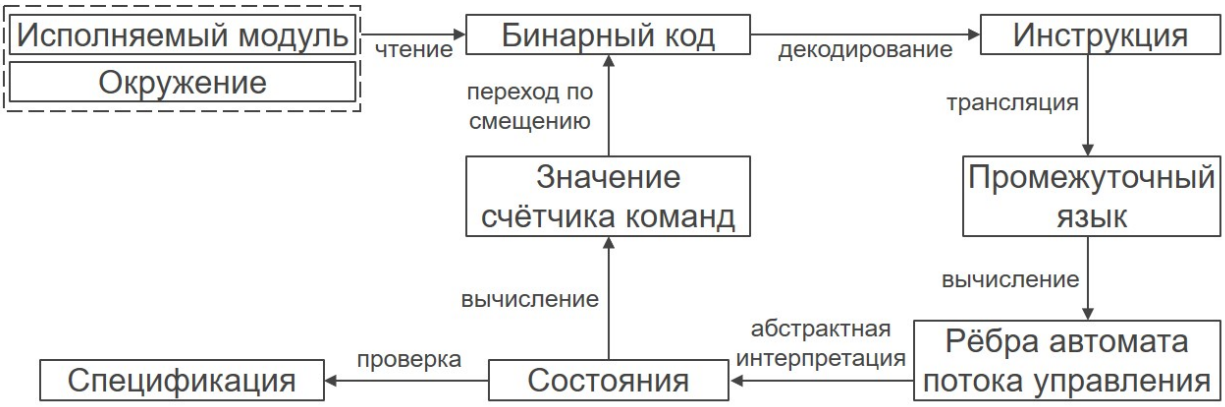


Рис. 2: Единая архитектура дизассемблирования и анализа Jakstab [8]

Подход Jakstab существенно отличается от подходов BAP, IDA Pro и других подобных инструментов: он не разделяет работу на последовательные стадии дизассемблирования и построения графа потока

[3]Register transfer language

[4]Граф, вершины которого помечены логическими состояниями, а рёбра — инструкциями. Логическое состояние — это набор, состоящий из счётчика команд, значений регистров, содержимого памяти и динамически выделенных объектов.

управления, а производит реконструкцию потока управления, на лету дизассемблируя достижимые инструкции (рис. 2). Дизассемблирование является итеративным: реконструкция выполняется не за один просмотр входного модуля, а за несколько итераций дизассемблера, на каждой из которых извлекается новая информация о возможных переходах. При построении графа потока управления анализируется и поток данных: именно это позволяет строить предположения о содержимом регистров и разрешать зависящие от них динамические переходы.

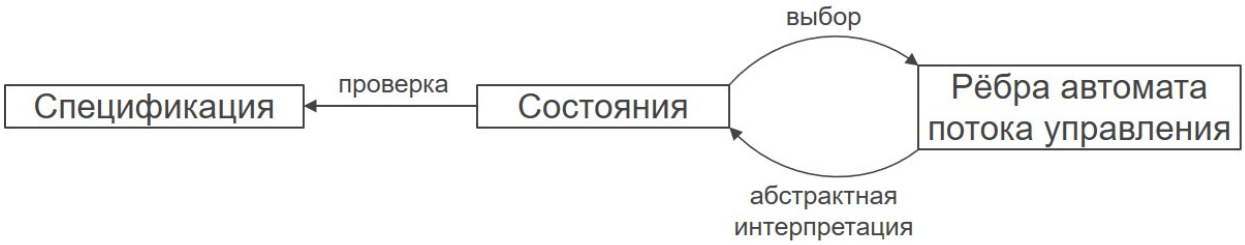


Рис. 3: Вторичный анализ на построенном автомате потока управления [8]

Jakstab позволяет использовать дополнительные анализы для построения графа потока управления (рис. 3), причём их можно свободно комбинировать. Затем —- после удаления мёртвого кода — можно отдельно запускать анализы на графе, полученном в результате основной работы, уже не изменяя его.

Для реализации анализов в Jakstab используется модифицированный фреймворк CPA (Configurable Program Analysis [3]), дополненный набором примитивов, подходящих для построения графа потока управления. Это позволяет реализовывать новые анализы, используя лишь специальные операторы фреймворка вместо написания всего алгоритма каждый раз с нуля. CPA используется как для анализа во время построения графа потока управления, так и для вторичного анализа.

Перечислим некоторые из используемых анализов:

- Ограниченное отслеживание адресов (Bounded Address Tracking) — основной анализ, используется по умолчанию;

- Распространение констант (Constant Propagation);
- Интервальный анализ (Interval Analysis) — полезен, например, для работы с конструкцией switch;
- Прямая подстановка выражений (Forward Expression Substitution) — распространение константных выражений, полезно для вычисления аргументов условных переходов;
- Анализ k-множествами (K-Set Analysis) — для каждой переменной собирает до k возможных значений на каждом смещении.

2.3. Сравнение платформ

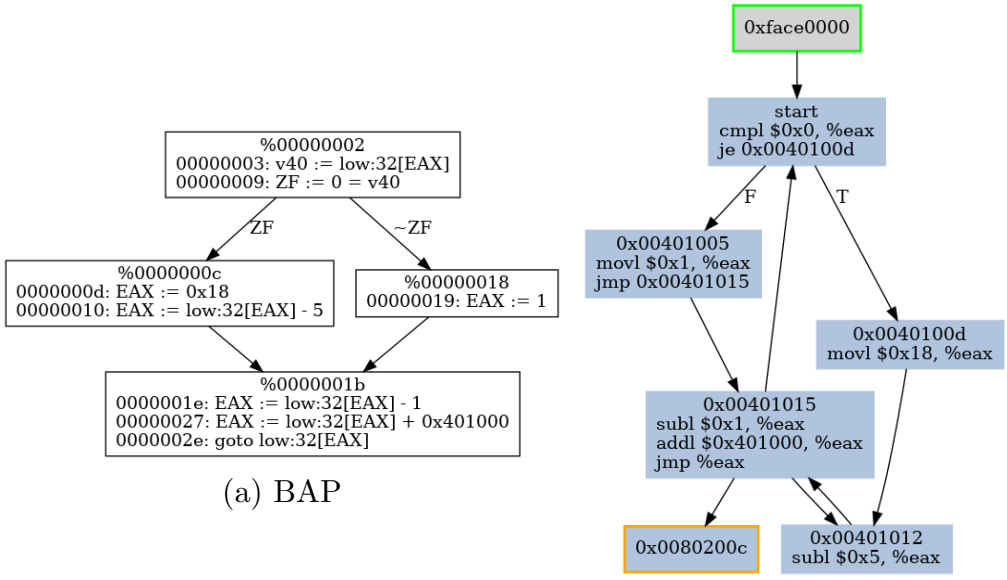
Теперь сравним BAP и Jakstab с целью выбора наиболее подходящей из этих платформ.

Продемонстрируем на небольшом примере (Listing 1) основную разницу результатов BAP и Jakstab. Для этого будем использовать бинарный модуль архитектуры x86, содержащий как прямые, так и косвенные переходы.

```
_start: cmp     eax, 0
        je      label1
        mov     eax, 1
        jmp     label2
        ret
label1:
        mov     eax, 24
        sub     eax, 5
label2:
        sub     eax, 1
        add     eax, _start
        jmp     eax
```

Listing 1: Пример бинарного модуля архитектуры x86

Из сгенерированных графов потока управления видно (рис. 4), что, как и ожидалось, с прямыми переходами справляются обе платформы. Однако косвенный переход по регистру `eax` игнорируется BAP'ом



(a) BAP

(b) Jakstab

Рис. 4: Графы потока управления, построенные разными платформами

и успешно разрешается Jakstab'ом. Благодаря анализу потока данных Jakstab определяет все возможные значения регистра `eax`, и проводит соответствующие рёбра в графе.

Как показывает этот и другие аналогичные эксперименты, Jakstab принципиально лучше, чем BAP, справляется с задачей разрешения косвенных переходов. И — опираясь на эксперименты, проведённые в [8], — можно утверждать, что с BAP нельзя добиться лучших результатов, даже используя плагин IDA Pro (который не подходит нам также по причинам, указанным ранее).

Таким образом, по результатам проведённого исследования, принято решение использовать Jakstab как основу для реализации статического анализа бинарных модулей z/OS.

3. Реализация

При реализации статического анализа использованы результаты работы Василия Мироновича [11], которому удалось встроить обработку бинарных модулей z/OS в платформу Jakstab. Разработанный прототип поддерживает ограниченное количество инструкций HLASM[5] и принимает файлы двух форматов — объектные файлы и загрузочные модули.

3.1. Графический интерфейс

В ходе работы был реализован графический интерфейс[6], как наиболее удобная оболочка для имеющегося интерфейса командной строки Jakstab.

Графический интерфейс — отдельное приложение, написанное на Java и принимающее путь к Jakstab в качестве одного из параметров. Пользователь задаёт путь к исполняемому модулю или объектному файлу, который надо проанализировать.

В приложении имеются две основные панели — панель управления и панель вывода. Панель вывода содержит три вкладки:

- Журнал служебной информацию — основные этапы анализа, ошибки, статистику (например, количество неразрешённых динамических переходов, количество рёбер в автомате потока управления) и другое;
- Граф и его краткое описание — путь к исходному модулю и время анализа. В эту вкладку можно вывести как граф потока управления, так и автомат потока управления (описанные в разделе Jakstab);
- Дизассемблированный код.

[5]High-Level Assembler — ассемблер фирмы IBM для z/Architecture.

[6]Графический интерфейс на GitHub: https://github.com/cthfvrl/jakstab-gui

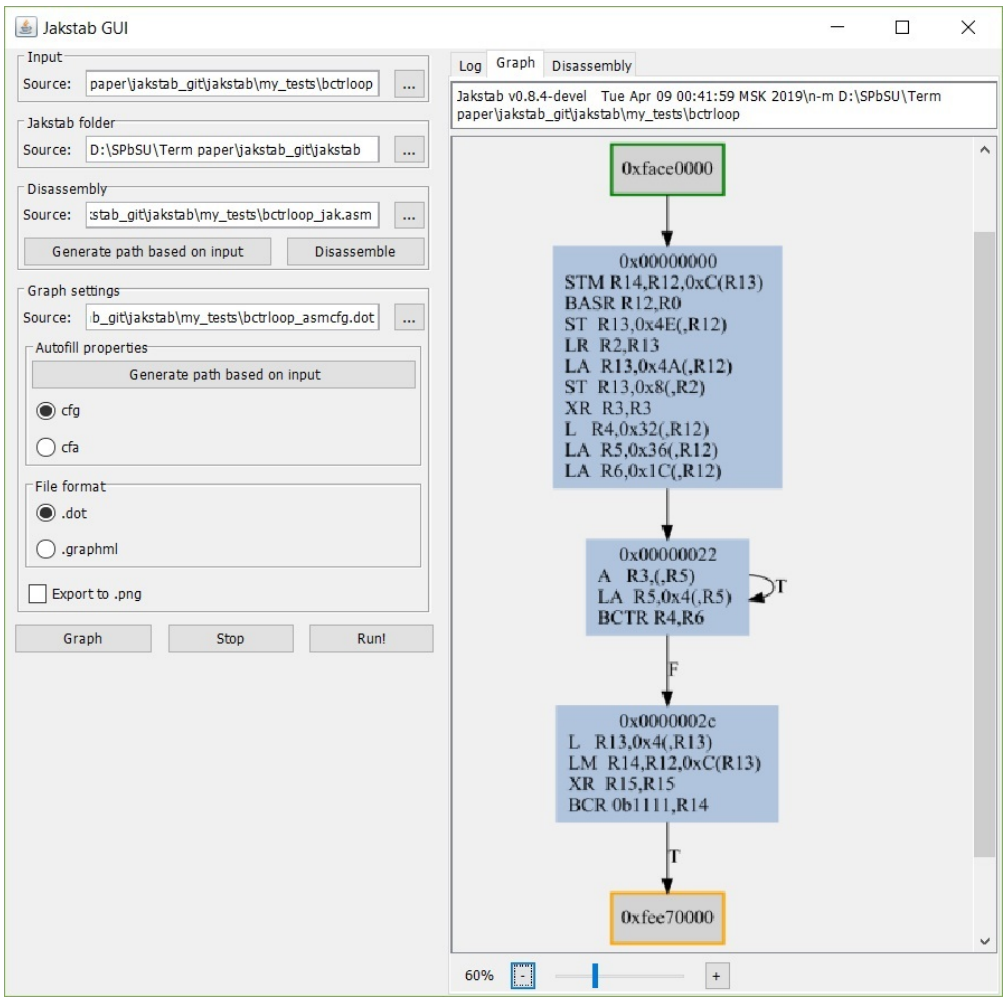


Рис. 5: Граф потока управления

На рис. 5 представлен пример результата работы программы. Выведен построенный Jakstab'ом (со встроенным модулем поддержки z/Architecture граф потока управления. На рис. 6 представлен результат дизассемблирования того же модуля. Отметим, что в данном примере есть динамический переход — `BCTR R4,R6` — и возможные адреса перехода вычислены верно.

На рис. 7 показана вкладка журнала после разбора другого модуля. В разделе статистики отражено, что в при анализе были разрешены не все динамические переходы (Unresolved Branches: 2).

В архитектуре Jakstab результаты работы сохраняются в том же месте, где и входной файл, причём имена новых файлов генерируются по известной схеме, поэтому в графическом приложении есть возможность заполнить эти поля — пути к графу и автомату потока управления и

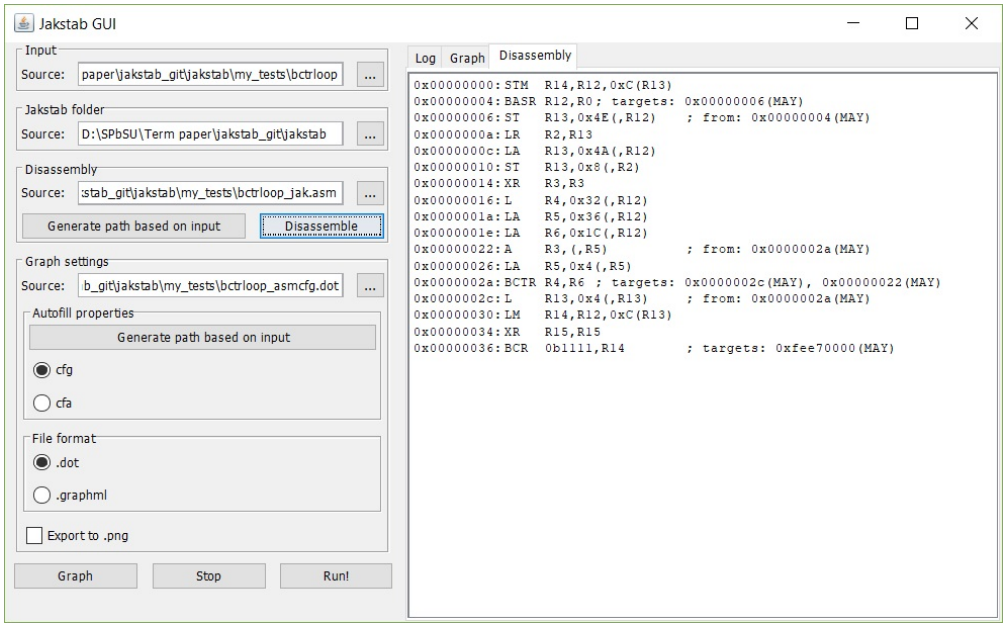


Рис. 6: Дизассемблирование

ассемблерному коду — автоматически. Также можно задать свои значения и открыть указанные файлы в тех же вкладках.

Графы в Jakstab можно генерировать в двух форматах: .dot и .graphml (это регулируется соответствующими флагами при запуске). Для формата .dot был найден плагин для Java graphviz-java [10], позволяющий генерировать изображение на основе .dot-файла. В графическом приложении этот плагин используется для отображения графа в соответствующей вкладке и для генерация картинок в формате .png.

Реализованы основные функции графического интерфейса, достаточные для запуска Jakstab с базовыми настройками и удобного отображения результата. В дальнейшем планируется добавление поддержки некоторых флагов Jakstab, доступных в интерфейсе командной строки, — в частности, флагов для выбора анализов, проводимых при реконструкции потока управления. Также можно добавить связь с мейнфреймом для получения исходных модулей напрямую, без предварительного скачивания пользователем.

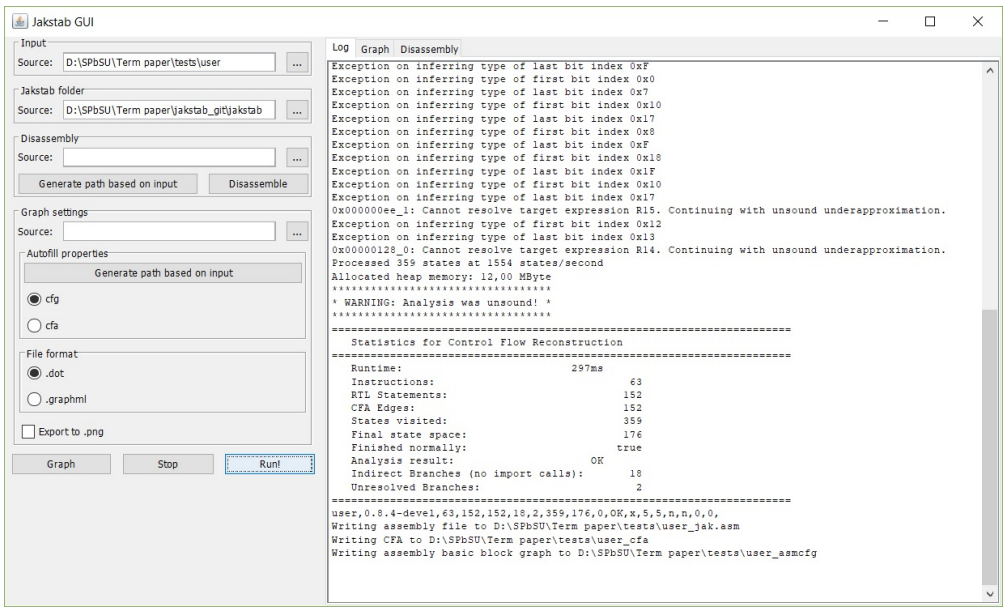


Рис. 7: Журнал

3.2. Статический анализатор

В данном разделе описывается развитие модуля поддержки z/Architecture[7].

3.2.1. Входные форматы

Имеющийся прототип поддерживает два вида файлов — объектные файлы[8] и загрузочные модули[9]. Они имеют открытый формат, описанный в [5].

Существует более новый формат программных объектов[10], пришедших на замену загрузочным модулям и имеющих ряд преимуществ[6]. Однако, в отличие от предшественника, этот формат не является открытым, и взаимодействие с программными объектами подразумевается только с использованием API линковщика (binder API). Поэтому в данной работе обработка такого формата не поддерживается.

С учётом этого было изменено определение формата исходного файла и добавлен вывод соответствующей ошибки в случае, если он не поддерживается.

[7]Код на GitHub: https://github.com/cthfvrl/jakstab

[8]Object module.

[9]Load module

[10]Program object.

3.2.2. Формат RI

Так как прототип поддерживает набор только наиболее важных инструкций, одной из задач является пополнение этого набора. В результате тестирования на рабочих исходных модулях были выявлены некоторые недостающие, но широко используемые инструкции.

Одной из них стала инструкция относительного перехода `BRAS` (синоним `JAS`). `BRAS` используется в довольно распространённом макросе вывода сообщений `WTO` (Listing 2).

```
                ...
79              WTO  'Reading...'
81+             CNOP  0,4
82+             BRAS  1,IHB0010A                BRANCH AROUND MESSAGE
83+             DC    AL2(14)                   TEXT LENGTH
84+             DC    B'0000000000000000'       MCSFLAGS
85+             DC    C'Reading...'             MESSAGE TEXT
86+IHB0010A     DS    0H
87+             SVC   35                        ISSUE SVC 35
                ...
```

Listing 2: Пример раскрытия макроса `WTO`

`BRAS` имеет формат `RI` [4], и обнаружилось, что он не был добавлен в прототип, так как ни одна инструкция такого формата ранее не входила в набор поддерживаемых инструкций. Для добавления нового формата в архитектуру необходимо сделать следующее:

- Добавить наследника `ZInstruction` — класс `ZRIInstruction`, определяющий тип операндов (в данном случае `ZRegister` и `Immediate`);
- Добавить наследника `ZOperandsDecoder` — класс `ZRIDecoder`, извлекающий операнды из потока байтов;
- Добавить в интерфейс `ZInstructionFactory` метод `newBranchInstructio` для создания инструкции перехода с определённым набором типов операндов, а также добавить имплементацию этого метода;
- Добавить в метод `decodeInstruction` класса `ZDisassembler` обработку нового формата;

- Добавить в класс `Translator` новый метод для перевода инструкций нового формата в набор предложений промежуточного языка RTL и внутри этого метода обработать конкретную инструкцию `BRAS`.

На рис. 8 приведён пример графа потока управления, построенного по модулю, содержащему макрос `WTO`. Инструкция `BRAS` (здесь она помечена синонимом `JAS`) успешно разобрана.

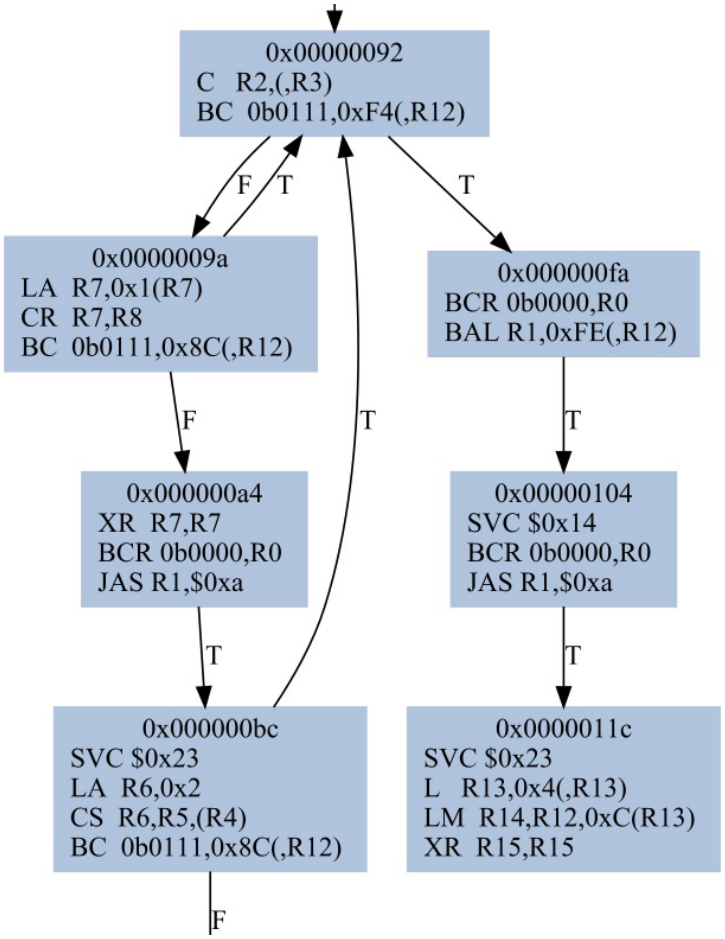


Рис. 8: Граф потока управления с инструкцией `JAS` (`BRAS`)

Трудоёмкость добавления новых форматов инструкций показывает необходимость дальнейшего рефакторинга архитектуры, который может выражаться в разработке собственного языка спецификации семантик, аналогичного используемому в Jakstab при переводе инструкций архитектуры x86 в RTL [8].

3.2.3. Инструкция EX

Одной из главных проблем остаётся инструкция `EX` — используемая на практике, но имеющая сложную динамическую семантику. Она берёт инструкцию по динамически определяемому адресу и исполняет её с частичными изменениями — второй байт инструкции логически складывается с содержимым регистра-операнда `EX`.

Определение положения исполняемой инструкции можно было бы имплементировать аналогично динамическим переходам. Однако изменение кода трудно описать в предложениях RTL. Это связано с тем, что семантика `EX` специфична для z/Architecture, а Jakstab создавался под ощутимым влиянием архитектуры x86.

Поведение `EX` можно сравнить с самоизменяющимся кодом (хотя сам модуль не меняется, логическое сложение происходит только при исполнении `EX` уже после прочтения целевой инструкции из памяти), разбор которого обычно не рассматривается при статическом анализе. Jakstab предполагает встраивание и такого вида анализа, но это представляет существенную сложность. Поэтому на данном этапе принято решение не поддерживать модули с инструкцией `EX`.

3.2.4. Обработка неподдерживаемых инструкций

В изначальной версии прототипа при достижении неподдерживаемой инструкции выводилась соответствующая ошибка и исполнение программы завершалось.

```
Error during disassembly: There is no instruction with such opcode in our
instruction set! Add it to file zArchitecture_formats_and_types.txt!
opcode = c0f4
Instruction could not be disassembled at: 0x00000034
ERROR: Replacing unknown instruction with HALT.
```

Рис. 9: Фрагмент вывода служебной информации

Теперь вместо этого инструкция заменяется на предложение промежуточного языка `HALT` (рис. 9), аналогично тому, как это сделано в реализации Jakstab для архитектуры x86. Это позволяет продолжить анализ, хотя и в ущерб точности.

3.3. Тестирование

Прототип модуля поддержки z/Architecture был простестирован ранее [11] на концептуально важных примерах. Некоторые из них:

- программа, состоящая только из пролога и эпилога, сохраняющая и восстанавливающая регистры;
- циклы (с использованием инструкций `BCT`, `BCTR`);
- условные переходы (`BC`, `BCR`);
- конструкция switch с использованием таблицы переходов.

С учётом изменений, описанных в разделе Статический анализатор, модуль продолжает корректно работать на данных примерах, генерируя достоверные графы потока управления. Замена неподдерживаемых инструкций на `HALT` позволила также запускать прототип на любых исходных модулях корректных форматов.

Помимо специальных примеров, прототип был протестирован на ряде рабочих программ — на некоторых системных модулях z/OS и модулях, выполняющих различные прикладные задачи. Результаты тестирования можно видеть в таблице 1.

Номер теста	1	2	3	4	5
Количество разобранных инструкций	190	63	63	20	17
Количество динамических переходов	54	18	14	4	3
Количество невычисленных переходов	8	2	2	1	0
Процент невычисленных переходов	14%	11%	14%	25%	0%

Таблица 1: Статистика тестирования

Заключение

При выполнении работы были достигнуты следующие результаты:

- Проведено исследование предметной области;
- Рассмотрены известные решения для других архитектур и выбрано наиболее подходящее для встраивания поддержки z/OS;
- Разработан графический интерфейс для Jakstab для более удобного запуска анализа и вывода его результатов;
- Произведена частичная доработка взятого за основу прототипа и тестирование добавленных улучшений.

Результаты работы представлены на ежегодной всероссийской научной конференции СПИСОК-2019.

Список литературы

[1] BAP: A Binary Analysis Platform / David Brumley, Ivan Jager, Thanassis Avgerinos, Edward J. Schwartz. — Springer, Berlin, Heidelberg, 2011. — Vol. 6806 of Lecture Notes in Computer Science.

[2] BAP: Binary Analysis Platform. — `https://github.com/BinaryAnalysisPlatform/bap/`. — Дата обращения: 16.05.2019.

[3] Beyer Dirk, Henzinger Thomas A., Théoduloz Grégory. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. — Springer, Berlin, Heidelberg, 2007. — Vol. 4590 of Lecture Notes in Computer Science.

[4] IBM Corporation. z/Architecture Principles of Operation, Tenth Edition. — 2012.

[5] IBM Corporation. z/OS MVS Program Management: Advanced Facilities, Version 2 Release 2. — 2015.

[6] IBM Corporation. z/OS MVS Program Management: User's Guide and Reference, Version 2 Release 2. — 2015.

[7] IDA. — `https://www.hex-rays.com/products/ida/`. — Дата обращения: 16.05.2019.

[8] Kinder Dipl.-Inf. Johannes. Static Analysis of x86 Executables : Ph. D. thesis / Dipl.-Inf. Johannes Kinder ; Technische Universität Darmstadt. — 2010.

[9] Kinder Johannes, Veith Helmut. Jakstab: A Static Analysis Platform for Binaries. — Springer, Berlin, Heidelberg, 2008. — Vol. 5123 of Lecture Notes in Computer Science.

[10] graphviz-java. — `https://github.com/nidi3/graphviz-java/`. — Дата обращения: 16.05.2019.

[11] Миронович В.С. Статический анализ бинарных модулей среды z/OS. — СПбГУ, 2017.