Повышение уровня представления трасс выполнения программ[1]

Назаров А. Г., Климушенкова М. А., Довгалюк П. М., Макаров В. А.,
Anton.Nazarov@ispras.ru, <Maria.Klimushenkova@ispras.ru>,
<Pavel.Dovgaluk@ispras.ru>, <Vladimir_Makarov@ispras.ru>

Аннотация. Использование динамического анализа для понимания поведения программы является распространенной практикой в наши дни. Одним из видов динамического анализа является анализ трасс выполнения программ. В то же время, трассы выполнения программ, состоящие из последовательности выполненных процессором инструкций, слишком велики для непосредственного восприятия человеком. Поэтому актуальной является задача повышения уровня представления таких трасс. В данной статье предлагается метод повышения уровня представления, предназначенный для построения модели алгоритма по трассе выполнения программы.

Ключевые слова. динамический анализ; восстановление переменных; восстановление интерфейсов; декомпиляция.

1. Введение

Динамический анализ – это анализ данных, полученных от выполняющейся программы. Он позволяет выявить те свойства программы, которые недоступны при анализе ее исходных текстов из-за их сложности, намеренной запутанности или позднего связывания. В случае, если исходные тексты программы недоступны, применяется динамический анализ бинарного кода.

Один из видов динамического анализа – анализ трасс выполнения программ. В случае выполнения анализа бинарного кода трасса представляет собой последовательность машинных инструкций в порядке их выполнения в программе. Но для того, чтобы понять работу алгоритмов, попавших в трассу, необходимо применять методы выделения из нее высокоуровневых сущностей и их фильтрации. Повышение уровня представления и фильтрация уменьшают объем обозреваемых данных и упрощают анализ алгоритма.

Динамический анализ трасс выполнения программ имеет как достоинства, так и недостатки, по сравнению со статическим анализом исполняемого кода. Недостаток заключается в том, что трасса не содержит инструкций из тех участков кода, которые не выполнялись во время снятия трассы. Достоинство

[1] Работа выполнена при поддержке РФФИ (грант 11-07-00353)

же трасс выполнения в том, что они содержат актуальные пути выполнения программы, а также информацию о значениях регистров.

Повышение уровня представления программы от бинарного кода до С-подобного языка называется декомпиляцией. Существует ряд реализаций, выполняющих декомпиляцию бинарного кода, например [[1]]. Однако, данные подходы не применялись в случае использования динамического анализа.

В данной статье предлагается метод повышения уровня представления трасс выполнения программ, основанный на выделении из них функций и структур данных. Предлагаемый подход использует уже достигнутые результаты по извлечению алгоритма из бинарного кода, описанные в [[2]]. Предполагается создание средства для анализа трасс выполнения программ, позволяющего просматривать их не в виде машинного кода, а в виде конструкций и переменных на С-подобном языке. Пользуясь таким представлением, можно строить высокоуровневую модель алгоритма.

2. Восстановление модели алгоритма

В данном разделе описывается общий подход к решению задачи восстановления модели алгоритма по трассам, и описываются задачи, решаемые авторами в рамках данного подхода. Под **моделью алгоритма** в конечном итоге понимается статическое высокоуровневое представление алгоритма, полученное в результате анализа трассы или множества трасс исполнения программы.

Реализация алгоритмов данного подхода выполняется в виде модулей для системы динамического анализа бинарного кода TrEx [[2]]. Общий комбинированный (статический и динамический) подход к анализу ПО с помощью данной системы и интерактивного дизассемблера IDA Pro описан в статье [[3]].

Общая схема предлагаемого подхода изображена в виде диаграммы потоков данных на рис. 1. Элементы диаграммы, нарисованные пунктирными линиями, находятся на стадии разработки и на момент написания статьи, не имеют реализации в системе TrEx.

2.1. Общее описание процесса восстановления алгоритма

На рис. 1 изображена последовательность выполняемых алгоритмов (вместе с их входными и выходными данными), составляющих предлагаемый процесс восстановления модели алгоритма. Чтобы описать эту последовательность понадобится использовать ряд терминов. Следующие понятия были подробно рассмотрены в других работах, касающихся среды TrEx [[2]].

Трасса (Trace) – трасса исполнения анализируемой программы, содержащая последовательность выполненных процессором инструкций и значения регистров на каждом таком шаге. Для увеличения покрытия кода

анализируемой программы может сниматься несколько трасс с различными входными данными.

Логическая модель функции (Function model) – формальная низкоуровневая модель функции (логический блок), которая может содержать списки входных и выходных логических параметров, описание зависимостей типа «вход-выход» между логическими параметрами, относительный адрес точки входа в функцию и относительные адреса возможных точек выхода функции. В процессе восстановления модели алгоритма необходимо выделять в трассе функции, имеющие отношение к исследуемому алгоритму, и описывать их формальные модели.

Логический параметр формальной модели (Logic parameter) – низкоуровневый элемент адресного пространства являющийся входом и/или выходом функции, описываемой соответствующей формальной моделью. В процессе восстановления модели алгоритма необходимо восстанавливать зависимости по данным между логическими параметрами и детализировать их, чтобы определить способ преобразования входов в выходы для заданной модели, то есть функцию преобразования значений входных параметров в значения выходных параметров.

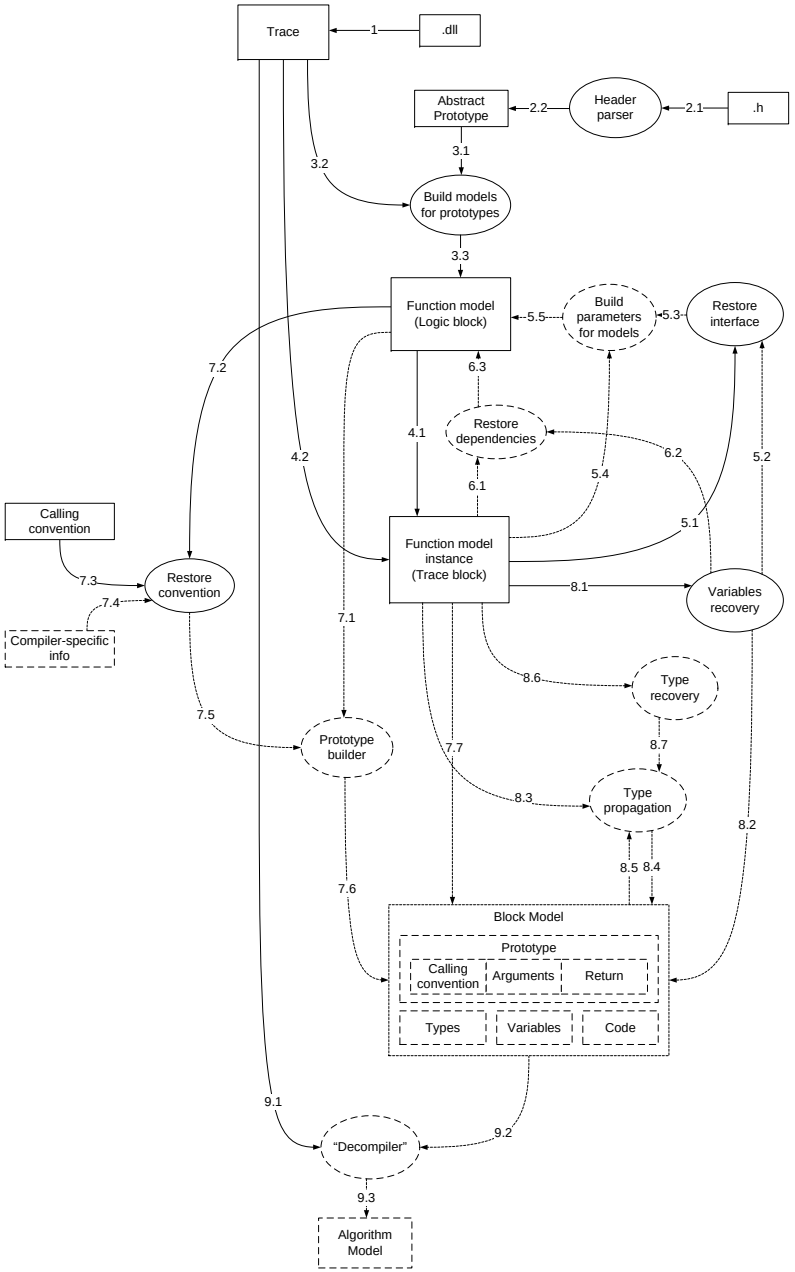

Рис. 1. Диаграмма потоков данных при автоматическом восстановлении модели алгоритма по трассе.

Блок трассы (Function model instance) – реализация логической модели функции, которая соответствует вызову данной функции в трассе и содержит списки реализаций входных и выходных логических параметров с восстановленными (по возможности) значениями при входе и выходе из блока соответственно, описание фактических зависимостей между входами и выходами, а также адрес начала и адрес конца блока в соответствующей трассе. В процессе восстановления модели алгоритма формальные модели строятся при наличии их реализаций (то есть последовательностей инструкций, соответствующих вызовам функции) в трассе (от блоков трассы для вызовов функций к формальным моделям функций). По соответствующим блокам трассы строятся логические параметры и восстанавливаются зависимости по данным между этими параметрами, а также по возможности восстанавливаются их значения.

Следующие понятия ранее в системе TrEx не использовались и вводятся впервые:

Абстрактный прототип (Abstract prototype) – высокоуровневый, абстрактный с точки зрения трассы, прототип функции, полученный в результате разбора заголовочного файла на языке C. Абстрактный прототип функции содержит: имя функции, тип возвращаемого значения, идентификатор используемого соглашения о вызовах (ключевое слово), типизированный список аргументов функции. В процессе восстановления модели алгоритма абстрактные прототипы известных библиотечных функций, вызовы которых присутствуют в трассе, извлекаются из соответствующих заголовочных файлов и являются источником достоверной информации о типах данных высокоуровневых аргументов функций.

Абстрактный аргумент функции (Abstract argument) – абстрактный с точки зрения трассы аргумент функции, полученный в результате разбора заголовочного файла и принадлежащий абстрактному прототипу функции. Абстрактный аргумент функции состоит из имени и типа.

Модель функционального блока (Block model) – высокоуровневое представление функционального блока (функции), содержащее в себе информацию о конкретном функциональном блоке анализируемого алгоритма, которая необходима для построения модели алгоритма в целом. Содержит в себе информацию из абстрактного прототипа (имя функции, типы данных аргументов, и др.), карту размещения абстрактных аргументов на логические параметры формальной модели, восстановленную информацию об используемых переменных (локальных и глобальных) и типах данных, а также высокоуровневое представление зависимостей между аргументами (входами функционального блока) и результатом (выходами функционального блока). В процессе восстановления модели алгоритма строятся модели всех функциональных блоков, являющихся частями исследуемого алгоритма, а также устанавливается характер взаимодействия между отдельными блоками.

Модель алгоритма (Algorithm model) – представление алгоритма в терминах языка программирования высокого уровня. Является конечным результатом анализа трассы и может быть передана пользователям, являющимися специалистами в предметной области, к которой относится работа алгоритма.

Также на схеме фигурирует **информация, специфичная для конкретного компилятора/платформы** (например, размеры встроенных типов данных). Она используется вместе с описаниями соглашений о вызовах в процессе разметки абстрактных аргументов функции на логические параметры формальной модели этой же функции. В рамках описываемого в данной статье подхода компилятор указывается пользователем вручную (используются только компиляторы языков C/C++). В статье [[4]] предлагается подход к идентификации компилятора по бинарному файлу программы, основывающийся на методах кластеризации идиоматических байтовых структур, получаемых на выходе различных компиляторов. При правильном развитии данного подхода его можно применить для анализа трасс выполнения программ.

На диаграмме (рис. 1) обозначены следующие алгоритмы:

- алгоритм идентификации интерфейса блока трассы (**Restore interface**), описанный ранее в [[5]];
- алгоритм восстановления соглашения о вызовах (**Restore convention**);
- алгоритм разбора заголовочных файлов для извлечения объявлений функций (**Header parser**), описанный более подробно в разделе 3;
- алгоритм построения формальных моделей для функций, которым поставлены в соответствие прототипы (**Build models for prototypes**);
- алгоритм построения логических параметров для формальных моделей (**Build parameters for models**);
- алгоритм восстановления зависимостей между логическими параметрами формальных моделей (**Restore dependencies**);
- алгоритм построения прототипа функции, то есть размещения аргументов абстрактного прототипа на низкоуровневые параметры (**Prototype builder**);
- алгоритм восстановления переменных (**Variables recovery**). Его задачей является выделение всех элементов памяти (ячеек памяти и регистров), которые используются в функции. Среди них выделяются входные и выходные элементы. Входными являются элементы, которые используются без инициализации внутри функции, к ним будут относиться параметры вызова, статические и динамические переменные. Выходными являются элементы, которые получили свое значение внутри функции, и используются вне функции, к

ним будут относиться возвращаемое значение функции, статические и динамические переменные. Более подробно алгоритм описан в разделе 4;

- алгоритм восстановления базовых типов данных (**Type recovery**). Размеры регистров/ячеек памяти, на которых размещаются переменные, а также семантика инструкций ассемблера накладывают ограничения на типы своих операндов. Анализируя эти ограничения, восстанавливаются типы некоторых переменных. Этот анализ позволяет получить информацию о типах переменных тех функций, для которых не известны прототипы;
- алгоритм распространения типов данных (**Type propagation**). Типы распространяются внутри функции и между функциями в трассе. Информация о типах передается между функциями с помощью параметров, возвращаемых значений и глобальных переменных;
- обобщённый алгоритм автоматического восстановления модели алгоритма по моделям функциональных блоков (**«Decompiler»**) представлен на диаграмме как некоторый промежуточный алгоритм построения модели алгоритма по моделям функциональных блоков, входящих в этот алгоритм (в процессе дальнейшей работы этот алгоритм будет разрабатываться более детально).

Решение общей задачи восстановления модели алгоритмов состоит из нескольких этапов. На диаграмме потоков данных потоки данных пронумерованы в соответствии с номерами этапов, которые описываются далее.

1. Извлечение символьной информации из бинарных файлов библиотек и привязка символов к соответствующим функциям в трассе.

2. Извлечение объявлений библиотечных функций из заголовочных файлов, построение абстрактных прототипов и их привязка к функциям в трассе (по имени из прототипа и соответствующему символу). При наличии перегруженных функций (актуально для C++) возникает проблема декорирования имён функций (name mangling). Для решения этой проблемы необходимо разработать и реализовать алгоритм выполнения операции обратной декорированию имени функции (name demangling). В результате выполнения такого алгоритма по декорированному имени функции (символу) можно получить объявление функции, по которому можно построить корректный абстрактный прототип и привязать его к соответствующей функции в трассе.

3. Построение логических моделей для функций в трассе, которым поставлен в соответствие абстрактный прототип. На данном этапе

модели не содержат информации о логических параметрах и о зависимостях между логическими параметрами.

4. Построение блоков трассы для логических моделей, построенных на предыдущем этапе, по вызовам соответствующих функций.

5. Для каждой логической модели функции выполняется восстановление интерфейса каждого блока трассы, соответствующего этой модели, и строятся её логические параметры.

6. Для каждой формальной модели выполняется восстановление зависимостей между её логическими параметрами.

7. Для каждой формальной модели при необходимости выполняется восстановление соглашения о вызовах (если не указано в абстрактном прототипе или не указано вручную пользователем), строится карта соответствия между аргументами абстрактного прототипа и логическими параметрами формальной модели функции и сохраняется в соответствующей модели функционального блока (**Block model**).

8. Восстановление переменных и типов данных для переменных, распространение типов (восстановленных и полученных из абстрактных прототипов функций) для функций, формальные модели которых построены на этапе 3. Все полученные данные по функциональным блокам сохраняются в соответствующих **Block model**.

9. Дальнейший анализ построенных моделей функциональных блоков для построения модели интересующего алгоритма.

3. Загрузка прототипов библиотечных функций

При снятии трассы выполнения некоторой анализируемой программы в трассу, кроме пользовательского кода, попадают вызовы функций различных библиотек (модулей). Описание таких функций, как правило, известно, то есть существует возможность получить их полные прототипы (то есть число параметров и их типы). Основная идея данного подхода заключается в подключении достоверной информации о типах данных, получаемой из описаний библиотечных функций, к трассе. В дальнейшем такую информацию можно распространить, используя методы распространения семантики данных (например, описанные в [[6]]). Таким образом, достоверная информация о типах данных может быть получена и в точках соприкосновения с кодом исследуемого алгоритма (например, вызовы библиотечных функций в пользовательском коде), что позволит выполнять анализ интересующих алгоритмов более высоком уровне представления кода.

3.1. Ограничения

В рамках предлагаемого подхода анализируются вызовы функций в трассе, соответствующие библиотечным функциям, то есть функции, для которых в

трассе присутствует символьная информация (имя функции или декорированное имя функции). Здесь речь идёт о функциях, чьи прототипы описаны в заголовочных файлах на языке C, и экспортируемых различными статическими или динамическими библиотеками. Соответственно система типов данных, используемая в разрабатываемых алгоритмах, основывается на системе типов языковой группы C.

Некоторые задачи (например, задача определения соглашения о вызовах для функции в трассе при отсутствии информации о соглашении в соответствующем заголовочном файле) на момент написания статьи не имеют автоматического решения, поэтому для их решения требуется интерактивный режим работы алгоритмов, позволяющий определять некоторые параметры вручную.

3.2. Реализация

Разбор заголовочных файлов – одна из задач, реализация которых требуется для внедрения предлагаемого подхода в среду TrEx. Абстрактные прототипы функций, извлеченные из заголовочных файлов, используются в качестве достоверного источника информации о типах данных параметров и возвращаемых значений.

Для преобразования заголовочных файлов из текстового формата в формат абстрактного синтаксического дерева используется свободная библиотека синтаксического анализа clang [[7]].

Полученное абстрактное синтаксическое дерево содержит в себе информацию о прототипах библиотечных функций: их имена, названия используемых соглашений о вызовах, названия и типы параметров, тип возвращаемого значения. Затем эта информация преобразуется в абстрактный прототип и сохраняется во внутреннем хранилище данных системы динамического анализа TrEx.

Таким образом, была выполнена реализация алгоритма загрузки прототипов функций. Она создает абстрактные прототипы, содержащие имена функций, а также имена и типы параметров. Реализация поддерживает работу с базовыми типами параметров и возвращаемых значений. В дальнейшем планируется расширение модуля для извлечения информации и о структурах данных, а также указателях на них.

4. Восстановление переменных

Алгоритм восстановления переменных был реализован в виде модуля к системе динамического анализа бинарного кода TrEx. Основными задачами модуля восстановления переменных являются:

- Разделение ограниченного числа регистров и повторно использующихся ячеек памяти на переменные
- Определение времени жизни переменных

- Соединение информации о переменных из разных вызовов одной и той же функции
- Определение области памяти, в которой находится переменная

Каждый процессор имеет определенный набор регистров и ограниченный объем памяти, а количество высокоуровневых переменных в программе не ограниченно [[8]], [[9]], [[10]]. Вследствие этого в одном регистре/ячейке памяти в разные моменты времени могут быть расположены разные переменные.

Прежде чем восстанавливать типы, необходимо выделить из низкоуровневых объектов (регистров/ячеек памяти) высокоуровневые объекты, представляющие собой переменные. В большинстве инструментов, которые занимаются восстановлением высокоуровневых объектов данных и их типов с помощью методов статического и динамического анализа, принципы такого разделения очень схожи. Первоначально переменные определяются в SSA форме [[11]], [[12]]. Это приводит к созданию большого количества переменных, с которыми сложно работать, поэтому их нужно объединить в логические объекты. В описываемом модуле переменные объединяются на основании утверждения, что модификация значения (изменение значения с участием старого значения) элемента памяти не является новым определением и не приводит к созданию новой переменной.

Другим подходом [[13]] является выделение переменной как компоненты связности графа определение-использование, построенного для всех используемых элементов. Такой подход преследует те же цели, что и описанный выше, то есть определение диапазона инструкций, когда ячейка памяти или регистр представляют собой одну логическую сущность.

Во время компиляции переменные языка Си могут быть размещены в статической памяти, на стеке, на регистрах и в динамической памяти. Куда в память компилятор разместит переменную зависит от ее времени жизни. Переменные с глобальным временем жизни, то есть глобальные переменные и локальные переменные со спецификатором static, размещаются в статической памяти. Память для них остается зарезервированной до конца выполнения программы. Переменные с локальным временем жизни, то есть локальные переменные и параметры функций, размещаются на стеке или на регистрах. Параметры могут передаваться на регистрах или в стеке. Для того, чтобы явно отделить параметры вызова и возвращаемое значение от глобальных переменных и сохраняемых регистров, нужно знать соглашение о вызовах, используемое компилятором.

В разработанном алгоритме под временем жизни переменной понимается диапазон шагов от создания переменной до ее последнего использования/изменения. Переменная создается в случае присваивания нового значения элементу памяти (регистру или ячейке памяти). Изменение,

то есть формирование нового значения с участием старого, не приводит к созданию новой переменной. Входные элементы функции определяются как элементы, использующиеся в функции, время жизни которых началось до вызова функции. Время жизни выходных элементов начинается до инструкции возврата из функции и заканчивается после.

В конструкциях ветвления выполнение программы может пойти по разным веткам. В каждой такой ветке одна и та же ячейка памяти или регистр может получать новое значение, и как следствие на основе этого будет созданы новые переменные. При соединении этих путей переменные объединяются.

В трассе может быть несколько проходов одной функции, а в разных вызовах могут быть различные пути выполнения. После анализа отдельных вызовов, вся информация объединяется, трассы вызовов накладываются друг на друга [[14]]. Чтобы объединить информацию о переменных нужно разделить память по типам. У статической переменной в каждом вызове будет один и тот же адрес. По одному адресу всегда находится одна переменная, но у нее могут быть разные типы, на пример, в случае с union. Стековые переменные будут иметь разный адрес, но постоянное смещение. У динамических переменных при каждом заходе может быть разный адрес, и по одному адресу могут располагаться разные переменные.

Модуль восстановления переменных предполагает плоскую модель памяти, где каждому процессу выделяется свое виртуальное адресное пространство, в которое загружается код приложения, секции данных, все необходимые библиотеки, а также выделяется области под стек и динамическую память. Загруженные в память библиотеки имеют свою статическую память, но пользуются общим стеком и кучей динамической памяти, хотя могут иметь и свой менеджер памяти.

Секции статических данных представляют собой непрерывные области памяти постоянного размера, которые содержат инициализированные и неинициализированные данные.

Область, отведенная под динамическую память, не является непрерывной, ее размер изменяется в зависимости от потребностей исполняемой программы. Так как динамическая память организована по принципу кучи, участки выделяемые менеджером памяти непоследовательны, а так же могут принадлежать разным областям (например, выше и ниже стека).

Стек представляет собой непрерывную область памяти, у которой есть начальный адрес ("дно" стека). Область стека не имеет определенного размера, она может увеличиваться или уменьшаться в зависимости от потребностей программы. Стек как правило "растет вниз", т.е. адрес следующей выделенной ячейки меньше адреса предыдущей, но может "расти вверх". Область стека может увеличиваться, пока есть свободное место, которое может быть занято растущей областью динамической памяти. В разные моменты времени одни и те же участки памяти могут занимать как стек, так и динамическая память [[8]], [[9]], [[10]].

Сложно отделить статическую память от динамической. Поскольку область статических данных является секцией загружаемого модуля, то можно по адресу загрузки модуля и его размеру определить принадлежит ли рассматриваемая ячейка статической памяти модуля. Где менеджер памяти может выделить место для динамических переменных понять сложно, поэтому необходимо находить функции управления памятью и выполнять анализ их использования.

Во время выполнения программы стек занимает область памяти от "дна" стека до текущего значения указателя стека. Потоки разделяют общее адресное пространство процесса, но каждый имеет свой стек. Так же режим ядра и режим пользователя имеют свой стек. В трассе можно определить наибольшее значение указателя стека в определенном потоке в режиме пользователя, и считать его "дном", тогда ячейки, адрес которых попадает в диапазон между этим значением и текущем значением указателя стека, точно находятся в стеке. Однако, этот способ не сможет идентифицировать все ячейки, если запись трассы началась не с начала выполнения программы.

При вызове функции в стек сохраняются адрес возврата, сохраняемые регистры (если это требуется), выделяется место для локальных переменных. При многократных вызовах функции указатель стека в начале функции может быть разным, и как следствие, адреса локальных переменных будут разными. Постоянным будет лишь смещение относительно начала stack frame. Поэтому элементы стека, использующиеся в функции, идентифицируются по смещению.

В других инструментах, восстанавливающих высокоуровневые объекты данных, глобальные статические переменные определяются либо по шаблонам обращения к памяти (обращение по абсолютному адресу) [[13]], [[15]], [[16]], либо с помощью анализа секций данных исполняемого файла [[17]]. Локальные переменные, как правило, определяются по обращению с использованием указателя стека или его копии. Для выделения участков динамической памяти, ищутся вызовы функций malloc и free.

5. Заключение

В статье был описан метод повышению уровня представления трасс выполнения программ для построения моделей алгоритмов. Данный метод включает в себя работу нескольких модулей. Модули для выделения локальных переменных, параметров функций, а также загрузки прототипов функций из заголовочных файлов были реализованы для среды динамического анализа бинарного кода TrEx.

В дальнейшем планируется реализовать остальные модули для системы TrEx, позволяющие повысить уровень представления трассы с машинных инструкций до С-подобного языка, такие как восстановление базовых и структурных типов данных, восстановление прототипов функций с привязкой

их к переменным, обнаруженным к трассе, а также восстановление высокоуровневых управляющих конструкций и арифметических выражений.

Полученное высокоуровневое представление позволит строить модель алгоритма программы, которую можно использовать для обнаружения закладок и недокументированных возможностей, поиска обратного алгоритма, а также восстановления алгоритма при отсутствии исходных текстов программы.

Список литературы

[1] C. Cifuentes and K. J. Gough. Decompilation of binary programs. Softw. Pract. Exper., 25(7):811–829, 1995. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.4380250706, дата обращения 31.10.2012

[2] В.А. Падарян, А.И. Гетьман, М.А. Соловьев. Программная среда для динамического анализа бинарного кода. Труды Института системного программирования РАН, том 17, 2009 г. Стр. 51-72

[3] А.Ю.Тихонов, А.И. Аветисян. Комбинированный (статический и динамический) анализ бинарного кода. Труды Института системного программирования РАН, том 22, 2012 г. Стр. 131-152.

[4] Nathan E. Rosenblum, Barton P. Miller, Xiaojin Zhu. Extracting compiler provenance from program binaries. Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. Toronto, Ontario, Canada, 2010, pp. 21-28

[5] А.Г. Назаров. Автоматическая идентификация интерфейса блока трассы в задаче восстановления модели алгоритма. **Вестник НовГУ выпуск №62. 2012 стр. 41 - 50**

[6] J. Caballero, N.M. Johnson, S.McCamant, and D. Song. Binary code extraction and interface identification for security applications. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, February 2010, 1-18 pp

[7] clang: a C language family frontend for LLVM. http://clang.llvm.org/, дата обращения 31.10.2012

[8] Intel® 64 and IA-32 Architectures Software Developer's Manual. http://download.intel.com/products/processor/manual/325462.pdf, дата обращения 31.10.2012

[9] ARM® v7-M Architecture Reference Manual. https://silver.arm.com/download/ARM_and_AMBA_Architecture/AR580-DC-11001-r0p0-02rel0/DDI0403D_arm_architecture_v7m_reference_manual_errata_markup_1_0.pdf, дата обращения 31.10.2012

[10] MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS64® Architecture. http://www.mips.com/auth/MD00083-2B-MIPS64INT-AFP-03.02.pdf, дата обращения 31.10.2012

[11] Климушенкова М.А. Методы восстановления объектов данных из бинарного кода. **Вестник НовГУ выпуск №62. 2012 стр. 51 - 57**

[12] Alan Mycroft. Type-based decompilation. Proceedings of European Symposium on Programming, Mar. 1999, pp. 208 – 223

[13] E. Dolgova and A. Chernov. Automatic reconstruction of data types in the decompilation problem. Programming and Computer Software 35, Mar. 2009, pp. 105 – 119

[14] JongHyup Lee, Thanassis Avgerinos and David Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. Proceedings of the Distributed System Security Symposium, NDSS 2011, pp. 1 – 18

[15] G. Balakrishnan and T. Reps. DIVINE: Discovering variables in executables. Proceedings of the International Conference on Verification Model Checking and Abstract Interpretation, Jan. 2007, pp. 1 - 28

[16] G. Balakrishnan. WYSINWYX: What You See Is Not What You eXecute. PhD thesis, Computer Science Department, University of Wisconsin at Madison, Aug. 2007, pp. 20 - 97

[17] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. Proceedings of the Network and Distributed System Security Symposium, 2010, pp. 1 - 18