Программная среда для динамического анализа бинарного кода

В.А. Падарян, А.И. Гетьман, М.А. Соловьев
{vartan, thorin, eyescream}@ispras.ru

Аннотация. В данной работе рассматривается среда TrEx, позволяющая выполнять динамический анализ защищенного бинарного кода. Преследуемой целью является получение описания интересующего алгоритма. Среда реализует оригинальную методику анализа и предоставляет пользователю развитый набор программных средств, объединенных в рамках единого графического интерфейса. Подробно рассматриваются некоторые особенности среды, такие как аритектурнонезависимое API для работы средств анализа, возможности свертки вызовов функций, расширение пользовательского интерфейса скриптовым языком.

1. Введение

В выполнении исследований программного обеспечения, оформленного в виде готового к работе бинарного кода, часто без исходных текстов, заинтересованы многие организации, решающие задачи сертификации ПО, а также проблемы отладки своих разработок или их совместимости с другими программами и системами. Всем им требуется проводить анализ бинарного кода различных программ, как по отдельности, так и в комплексе со всей средой исполнения компьютера, иногда вплоть до самого низкоуровневого кода операционной системы. Целью таких исследований является получение информации об особенностях реализации алгоритмов, восстановление реализации этих алгоритмов и их представление в понятном аналитику виде, восстановление протоколов обмена информацией и форматов данных, а также поиск «недокументированных» возможностей, ошибок и уязвимостей.

В качестве примера рассмотрим ситуацию, когда требуется исследовать работу ПО, передающего данные по сети. Поскольку в работу вовлекается код не только самой программы, но и операционной системы и драйверов сетевых интерфейсов, недостаточно исследовать содержимое исполняемого файла программы. Требуется провести анализ всего стека сетевых протоколов, поскольку ошибки реализации и недокументированные возможности одних компонент могут влиять на работу и использоваться другими компонентами. Общий объём бинарного кода, который представляет для аналитика интерес,

может составлять десятки мегабайт. Решение этой задачи логично разбивается на решение ряда подзадач:

1. Поиск «точек зацепления», т.е. мест в системе, с которых целесообразно начинать исследование.
2. Раскрутка от точки зацепления назад (исследование кода, который привёл в точку) или вперёд (исследование кода, который работал после точки) с целью нахождения реализации алгоритма.
3. Поиск и анализ данных, влияющих на алгоритм по входу, и данных, образующихся по выходу алгоритма.
4. Восстановление алгоритмов, протоколов и форматов данных.
5. Выявление недекларированных возможностей, уязвимостей реализации.

Важно отметить, что перечисленные подзадачи возникают при решении практически любых других задач, связанных с исследованием бинарного кода. Главным (и, возможно, единственным до сегодняшнего дня) эффективным методом решения таких задач является комбинация методов статического анализа (используется дизассемблер и декомпилятор) и «ручного» динамического анализа (используется отладчик и некоторые вспомогательные средства – дамперы, мониторы и т.д.).

Статический анализ позволяет выполнять локальные исследования бинарного кода, к которому не применялись способы затруднения анализа. Из способов затруднения анализа отметим следующие:

- использование свойств процессорной архитектуры фон Неймана, в которой исполняемый код и данные без наличия специальной информации не различимы (и, соответственно, задача их различения в таком случае является алгоритмически неразрешимой);
- использование необходимости знания состояния и предыстории возникновения этого состояния вычислительной среды;
- использование «размазывания» кода алгоритма по всему исполняемому модулю или даже нескольким модулям.
- обычные методы затруднения статического анализа (например, переходы по вычисляемым адресам, смешанное кодирование инструкций, когда в одной инструкции может быть закодирована другая, которая может выполниться вместо основной при получении управления);
- упаковка исполняемого кода;
- обфускация (запутывающие преобразования);
- виртуальные машины.

Все перечисленные методы и особенно их комбинации при качественной реализации делают статический анализ совершенно неэффективным вследствие значительной трудоемкости, в результате чего аналитик вынужден применять динамические средства анализа (т.е. отладчик). Процесс отладки

более сложен и требует большей квалификации от аналитика, однако позволяет преодолевать некоторые проблемы, не решаемые в статике. Например, аналитик может посмотреть в отладчике, где в интересующий его момент функционирования системы находится код, а где данные, куда и в какой-то момент осуществляется переход по вычисляемому адресу, как выглядит распакованный код. Помимо того, аналитик может в некоторых случаях исследовать алгоритм распаковки, если этот алгоритм будет достаточно компактен и не защищен. Главной проблемой динамического анализа является то, что на данный момент нет доступных средств автоматизации труда, и практически все действия, выполняемые аналитиком, являются ручными операциями. Ситуацию усугубляет применение активных защит от динамического анализа, когда код содержит средства обнаружения работы под отладчиком и реагирования в виде неправильного функционирования, обфускация кода, использование виртуальных машин и многое другое. При правильной реализации такие методы защиты сводят к минимуму вероятность успеха исследования.

Вычислительная мощь компьютеров растёт с каждым днём, и это позволяет реализовывать всё более сложные, комплексные и ресурсоёмкие алгоритмы усложнения и запутывания кода. Человеческих сил и возможностей уже недостаточно для анализа гигантских объёмов информации, содержащихся в исследуемых системах. Развитие средств запутывания алгоритмов, в том числе, и на аппаратном уровне, влечет невозможность решения задач восстановления алгоритмов существующими методами. Средства запутывания все более востребованы на рынке, и сейчас наблюдается активное их развитие и распространение.

В данной работе описывается TrEx – программная среда динамического анализа бинарного кода. Возможности среды позволяют решать задачу восстановления алгоритма, преодолевая при этом комплекс средств защиты от анализа. Программные инструменты среды базируются анализе потоков данных в трассе выполнения программы и позволяют выполнять быстрое прототипирование специфических для каждого отдельного случая алгоритмов.

Статья состоит из пяти разделов. Во втором разделе кратко описывается методика, на основе которой предлагается решать поставленную задачу. В третьем разделе описывается программная система, реализующая эту методику. В четвертом разделе рассматриваются дальнейшие пути развития программной системы. В последнем, пятом, разделе делаются итоговые заключения.

2. Методика анализа бинарного кода

Среда TrEx реализует методику анализа, подробное рассмотрение которой можно найти в работе [1]. Здесь приводится краткое ее описание.

Исследуемая программа выполняется на симуляторе, обеспечивающем потактовую симуляцию инструкций, например, AMD SimNow [2], Virtutech Simics [3] и т.п. Аналитик добивается выполнения исследуемой функциональности на соответствующих начальных данных, и в это время осуществляется сохранение трассы. Трасса представляет собой непрерывную последовательность выполняемых на процессоре инструкций и снимки внутреннего состояния процессора при выполнении каждой инструкции. Таким образом, в трассе содержится значительный массив информации, описывающий все аспекты функционирования исследуемой программы (как заданных до начала трассировки, так и возникших в момент трассировки, например, пришедшие в систему сетевые пакеты). Трассировке практически не мешают никакие из существующих методов защиты исполняемого кода от анализа, потому что эти методы защищают только форму представления защищаемого кода, тогда как функциональность не может быть искажена. В частности, применение обфускации «диспетчер» [4] становится бесполезным, так как базовые блоки программы все равно должны выполняться в определенном порядке, и этот порядок непосредственно отражается в трассе.

Поскольку трасса содержит все состояния процессора, некоторая часть записей в ней относится к выполнению других процессов и коду самой операционной системы. После того как из трассы выделены инструкции, относящиеся к исследуемой программе, ищется место ввода начальных данных соответствующего алгоритма или место вывода результата его работы. Фиксируются те ячейки памяти или регистры, в которых эти данные расположены.

Далее из трассы выделяются только те инструкции, входных операндов которых достигли начальные данные (в случае фиксирования в трассе ввода) или работа которых повлияла на результирующие значения в выходе алгоритма (в случае, когда определялся выход алгоритма).

Данный анализ не является статическим, он представляет собой post-mortem обработку отладочных данных, но преследуемые им цели (лучшее понимание программы) аналогичны целям программного слайсинга. В дальнейшем этот способ фильтрации шагов трассы будем называть слайсингом трассы.

Полученный слайс программы содержит значительно меньшее количество инструкций и уже, как правило, является обозримым. Сокращение размеров трассы составляет, как правило, 3-4 порядка.

Следующим этапом идет построение работоспособного ассемблерного листинга. Из элементов трассы извлекаются инструкции, упорядочиваются по их расположению в памяти, для константных переходов и адресов памяти генерируются метки, строятся пролог и эпилог, обеспечивающие размещение и выдачу начальных и результирующих данных. Следует отметить, что нерешенной на данный момент остается проблема построения листинга для самомодифицирующегося кода.

Ассемблерная программа рассматривается как самодостаточный контрольный пример, выполнение которого способно подтвердить корректность всех проведенных аналитиком операций, поскольку его выполнение должно выдавать те же результаты, что были получены при работе исходной программы. Перед передачей прикладному аналитику контрольный пример может быть подвергнут декомпиляции в язык высокого уровня.

3. Среда TrEx

Общая схема устройства среды TrEx представлена на рис. 1. Пользователь взаимодействует со средой через графический интерфейс, предоставляя на входе трассу и получая на выходе ассемблерный листинг интересующего его алгоритма. Модули, принципиально обязанные учитывать специфику процессорной архитектуры, на которой выполняется программа, выделены жирным шрифтом. Эти компоненты позволяют располагать внутренним представлением, предоставляющим достаточное количество информации для работы средств анализа. При этом большинство из них не требуют специфических знаний об архитектуре. Даже если такие требования и возникают, например, в случае построителя листинга, происходит декомпозиция на базовый класс и набор производных классов, реализующих вспомогательные методы, где и расположено знание о семантике инструкций и выводных форматах ассемблера.

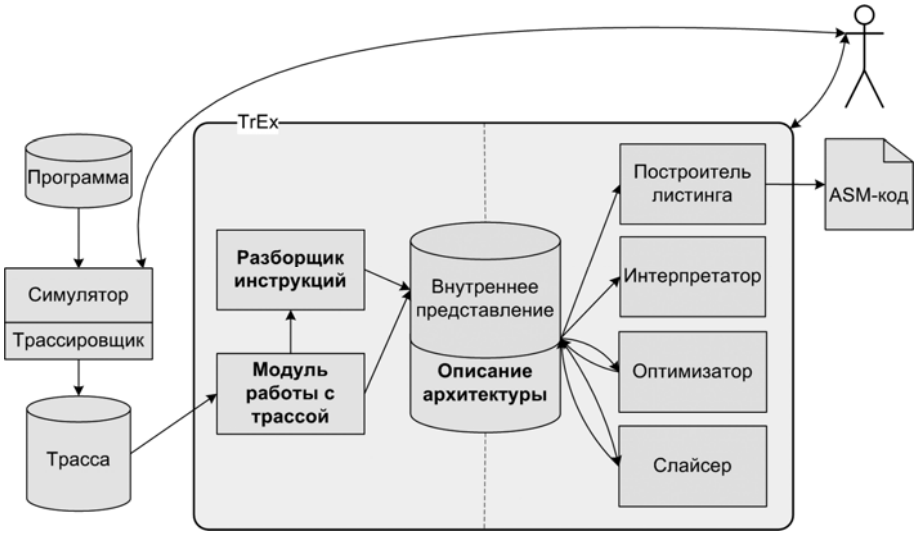

Рис. 1. Общая схема работы среды TrEx.

Далее будут рассмотрены некоторые особенности среды, такие как: аритектурнонезависимое API для работы средств анализа, возможности свертки вызовов функций, пользовательский интерфейс на основе скриптового языка.

3.1. Модель процессора общего назначения

Одним из поставленных требований было не закладывать в основы среды привязки к какой-либо процессорной архитектуре. Добиться выполнения этого требования удалось благодаря разработке Common API – библиотеки, предоставляющий архитектурнонезависимый интерфейс к трассировочным данным. В нем можно выделить три раздела, обеспечивающих доступ к описанию ресурсов вычислительной платформы, данным, сохраненным в трассе, и результатам разбора машинных инструкций.

Для описания регистрового файла, портов ввода-вывода и оперативной памяти машины в единообразном виде вводится модель адресных пространств. С точки зрения вводимой модели описываемая процессорная архитектура представляет собой набор адресных пространств, каждое из которых имеет свою разрядность адреса (рассматриваются разрядности в 8, 16, 32 и 64 бита). Элементами этих адресных пространств называются последовательные блоки, описываемые начальным адресом (с битовой частью) и длиной (также с битовой частью). Над элементами определены операции:

- `ElementsIntersect`. Проверка, имеют ли два указанных элемента пересечение как соответствующие области своих адресных пространств. При этом предполагается, что элементы различных адресных пространств пересечений не имеют.
- `CompareElements`. Сравнение элементов: для пары элементов e1; e2 указывается отношение между ними "<", ">" или "=". Конкретных требований к способу сравнения не предъявляется, однако данная операция обязана задавать полное отношение порядка на множестве элементов.

Обе операции можно задать, ограничившись лишь знанием об адресах и размерах рассматриваемых элементов. Однако в общем случае из соображений производительности может потребоваться для некоторых часто используемых элементов (таких, как, например, регистры) использовать статически заданные или однократно заполняемые таблицы.

С этой позиции оказывается удобным немного усложнить модель, введя понятие именованных элементов. Для каждого адресного пространства определяется множество (возможно, пустое) имен, с каждым из которых сопоставляется некоторый элемент этого адресного пространства. Элементы, имеющие имена, и будем называть именованными. Стоит отметить, что, несмотря на принципиальную возможность использования в рамках одного адресного пространства элементов с именами и без них, на практике такой необходимости не возникает. Причина заключается в том, что потребность в

присвоении имен обнаружилась только в отношении регистров, а для них на рассмотренных процессорных архитектурах не подразумевается произвольный доступ к части регистра.

Для поддержки именованных элементов необходимо ввести операции их перечисления (`EnumerateNamedElements`) и получения описания элемента по его имени (`GetNamedElementDescription`).

Предложенная модель позволяет единообразно описывать вершины графа зависимостей по данным, порождаемого выполнением инструкций трассы. В то же время сохраняется информация о виде элемента данных (регистр, порт ввода-вывода, память), которая может быть полезна при проведении анализа с более детальным учетом семантики инструкций трассы. Помимо того, предложенный подход позволяет описывать «вложенные» регистры, присутствующие в архитектуре INTEL IA-32.

Модель является максимально простой. В частности, она в явном виде не предусматривает наличие нескольких наборов регистров (регистровые окна в SPARC или теневые наборы регистров в некоторых реализациях архитектуры MIPS) или нескольких параллельно существующих пространств виртуальной памяти. Однако поддержку подобных особенностей можно сделать на уровне декомпозиции инструкций на зависимости по данным. Так, для поддержки теневых наборов регистров достаточно указать в адресном пространстве регистров регистры из всех таких наборов, а при разборе инструкции отображать регистры в смысле MIPS на регистры модели из конкретного теневого набора.

Описав архитектуру целевой машины в терминах адресных пространств, мы получаем возможность единообразно задавать всевозможные элементы данных, с которыми работает эта машина. Как было указано выше, применительно к системе TREX трасса состоит из шагов, для каждого из которых указана выполнявшаяся инструкция и значения некоторого подмножества регистров машины. Вместо того, чтобы предлагать единый формат трассы для всех архитектур, можно ограничиться парой операций, определенных над шагом трассы (их реализация уже будет зависеть от целевого процессора и выбранного формата хранения трассы):

- `GetInstruction`. Сообщает, какая инструкция выполнялась на указанном шаге. Инструкция записана в бинарном виде (строка байтов) и требует декодирования. Кроме того, для процессорных архитектур с переменной дли-ной инструкции в силу особенностей проведения трассировки фактическая длина инструкции может также быть неизвестной до проведения декодирования. В последнем случае длина полагается равной максимально возможной для данного процессора (например, 15 для INTEL 64).
- `GetItem`. По указанному элементу модели адресных пространств целевой архитектуры сообщает его значение на

данном шаге трассы перед выполнением соответствующей инструкции. Значение может оказаться неизвестным (например, значение запрошенного регистра вообще не трассировалось), в этом случае возвращается специальный признак «значение не-известно».

Легко видеть, что предлагаемый способ представления шагов трассы как интерфейс взаимодействия удовлетворяет требованиям:

- адекватности и полноты – указанный способ представления шагов трассы позволяет получать всю необходимую для работы системы информацию;
- минимальности – операции независимы и не выражаются друг через друга;
- простоты – нельзя представить предложенные операции в виде более мелких.

В качестве исходных данных для разбора инструкции выступают:

1. Сама эта инструкция в бинарном виде. Может быть получена из шага трассы (операция `GetInstruction`, описанная выше) или из отпечатка памяти.
2. Интерфейс для запроса значений регистров на момент выполнения инструкции. В случае работы с трассой этот интерфейс является прямым отображением на метод `GetItem` шага трассы, а в случае работы с памятью значения регистров полагаются неизвестными.
3. Информация о режиме работы процессора. Так как некоторые процессорные архитектуры (например, INTEL IA-32) способны функционировать в нескольких режимах, поведение инструкций в которых отличается (за счет различных механизмов отображения памяти, разрядности операндов и т. д.), необходимо явно указывать, в каком режиме работы выполнялась инструкция. Разметка шагов трассы по режимам сохраняется параллельно с трассой на этапе ее сбора.
4. Флаги декомпозиции на зависимости. На практике возникает необходимость исключать из рассмотрения при проведении слайсинга определенные категории зависимостей. Всего определено четыре таких категории.
   4.1 ADDRESS. Зависимости по вычисленному адресу. Например, в инструкции INTEL IA-32 `MOV EAX, [ECX+ESI*2-1]` при включенном отслеживании адресных зависимостей необходимо во множество рассматриваемых элементов добавить регистры ECX, ESI и DS (сегментный регистр данных).
   4.2 STACK. Зависимости по вычисленным адресам в стеке. При выключении этого флага адресные зависимости, соответствующие локальным переменным в стеке (в случае

INTEL IA-32 это те адреса, в которых фигурирует регистр SP/ESP) не будут генерироваться.

4.3 FLAG. Зависимости по флагам. Если эта опция выключена, флаги процессора при декомпозиции на зависимости не учитываются.

4.4 CF. Зависимости по управлению. Этот флаг определяет, следует ли учитывать при декомпозиции регистр или регистры, являющиеся счетчиком инструкций (для INTEL IA-32 это CS и IP/EIP).

Флаги указываются аналитиком при проведении слайсинга. В совокупности указанные данные представляют собой задание на разбор, которое передается в компонент разборщика. Результатом являются блок информации об инструкции, в котором содержится в общем виде информация об инструкции и ее операндах; производится поверхностная классификация инструкций по признаку отношения к передаче управления и операндов по их типу:

1. Мнемоника инструкции.
2. Число операндов и информация о каждом из них в отдельности, включающая следующие пункты.
   2.1 Классификация операнда в наиболее общем виде: константа; регистр или прямо указанный элемент памяти; элемент памяти, адресуемый косвенно.
   2.2 Для операндов-констант предоставляется возможность запроса их значения.
   2.3 Для операндов-регистров и ячеек памяти предоставляется возможность запроса соответствующих элементов в смысле модели адресных пространств. В случае косвенной адресации потребуется вычисление адреса, для чего используется переданная в задании ссылка на интерфейс запроса значений регистров.
3. Классификация инструкции по признаку отношения к передаче управления. Инструкции делятся на классы: инструкции вызова, инструкции возврата, инструкции безусловной передачи управления, инструкции условной передачи управления, остальные инструкции. Следует отметить, что знания мнемоники инструкции часто оказывается недостаточно для того, чтобы отнести ее к определенному классу. Так, например, инструкция безусловного перехода по адресу, записанному в регистре, JR в архитектуре MIPS должна быть отнесена в общем виде к инструкциям безусловной передачи управления. В то же время, ее форма `JR $31`, согласно принятым соглашениям, используется только для возврата из подпрограмм, так что в итоге принадлежность к определенному классу зависит от операнда инструкции.
4. Проверка, образуют ли две указанные инструкции пару «вызов-возврат».

Помимо того, требуется предоставлять информацию о зависимостях по данным, необходимую для работы алгоритма слайсинга. Декомпозиция инструкций производится на наборы троек ⟨t, I, O⟩, где t - тип зависимости, I - множество входных элементов адресных пространств, а O - множество выходных элементов адресных пространств, фигурирующих в данной зависимости.

Выделены следующие 5 типов зависимостей:

1. CHECK. Зависимость по управлению внутри инструкции (которая с точки зрения слайсинга все равно является зависимостью по данным).
2. GET. Зависимость по косвенному адресу (чтение из памяти). В качестве примера можно привести зависимость от регистров DS, ESI в инструкции `LODSB`.
3. KILL. Не являясь зависимостью как таковой, позволяет указать алгоритму слайсинга на тот факт, что во время выполнения инструкции некоторый элемент был уничтожен (т. е. его значение после выполнения инструкции непредсказуемо). Подобные ситуации характерны для флагов процессора INTEL IA-32.
4. SET. Зависимость по косвенному адресу (запись в память), например зависимость от ES, EDI в инструкции `STOSB`.
5. UPDATE. Все остальные зависимости по данным, не обладающие специальными свойствами.

Для описания предлагаемым образом некоторых инструкций (например, инструкции обмена XCHG процессора INTEL IA-32) необходимо использовать дополнительные теневые регистры, для которых определяется специальное адресное пространство, автоматически добавляемое ко всем моделям адресных пространств конкретных процессорных архитектур. Данное адресное пространство включает в себя 8 теневых регистров (S0-S7), каждый из которых может использоваться как 128-, 64-, 32-, 16- или 8-битный. Кроме того, определен один теневой битовый регистр `SU`.

После введения таких регистров инструкция `XCHG` распадается на три тройки:
$\{\langle U, \{o_0\}, \{S_7\}\rangle, \langle U, \{o_1\}, \{o_0\}\rangle, \langle U; \{S_7\}, \{o_1\}\rangle\}$, где $o_{\{0,\ 1\}}$ – операнды инструкции.

Можно видеть, что такая запись удовлетворяет требованиям к исходным данным алгоритма слайсинга (т. к. является одной из возможных форм записи графа зависимостей по данным). Преимуществом перед графовым представлением является то, что можно работать с отдельными такими тройками, указывая, таким образом, какая именно из содержащихся в инструкции зависимостей по данным повлекла включение инструкции в слайс.

В рамках описанного способа представления целевой процессорной архитектуры, для поддержки архитектур INTEL IA-32 и MIPS64 были

реализованы интерфейсы, соответствующие модели адресных пространств, подсистеме работы с трассой и разбору инструкций.

3.2. Свертка функций

Вследствие большого количества инструкций в трассе, аналитик перед началом анализа некоторым образом разбивает трассу на блоки (примером такого блока может служить вызов функции), а затем анализирует их по очереди. Таким образом, после завершения анализа некоторого блока и его описания, аналитику, как правило, не требуется работать с инструкциями этого блока. Возникает задача скрыть от аналитика проанализированные и описанные блоки кода для упрощения анализа других блоков. Для этого добавлена возможность создания визуальных свёрток. Этот механизм аналогичен свёрткам в визуальных средах разработки высокоуровневых языков программирования, например в редакторе MS Visual Studio для каждой функции существует возможность скрыть её тело, если оно в данный момент пользователя не интересует. В трассах это позволяет пользователю скрывать проанализированные фрагменты кода. Свёртка представляет из себя именованную блок последовательных инструкций. При создании свёртки (может происходить вручную или автоматически, как результат работы алгоритмов анализа структуры трассы) указывается имя свёртки и её границы. Свёртка может находиться в 2х состояниях:

- свёрнутом – при этом отображается значок «+» и имя свёртки, заданное при создании.
- развёрнутом – при этом отображается значок «-» на начальной строке блока и все инструкции, входящие в свёртку.

Свёртки могут быть вложенными, но не могут пересекаться.

При переходе на заданный шаг в трассе, в случае если он находится внутри свёртки и скрыт от пользователя, происходит автоматическое разворачивание этой свёртки, а также всех других свёрток, в которые она вложена.

Класс свёртки `CVisualFurl` представляет собой структуру из трёх полей: имя, начальный и конечный шаги свёртки и реализует методы для доступа к их значениям.

Класс `FurlManager` инкапсулирует всю работу со свёртками, отвечает за их сохранение и загрузку из файла. Так как объём трасс достаточно велик, хранить текстовое представление трассы невыгодно, поэтому при необходимости отобразить некоторый шаг трассы компонент отображения должен заново получать его текстовое представление. Компонент отображения обращается с запросами к `FurlManager`, для того чтобы правильно сгенерировать очередную строку в трассе

Опишем основные методы класса `FurlManager`, а ниже вкратце опишем его взаимодействие с компонентом отображения.

Основные методы класса `FurlManager`:

- bool isDirty() – возвращает состояние менеджера, изменялось состояние свёрток с последнего сохранения (требуется повторное сохранение) или нет;
- unsigned __int64 getStepsCount() – возвращает количество видимых шагов в трассе;
- bool addFurl(CVisualFurl furl) – добавить заданную свёртку;
- void delFurl(TracePosition visualPos) – удалить свёртку с началом в заданном шаге
- bool expandFurl(TracePosition visualPos) – свернуть/развернуть свёртку (изменить состояние) заданной начальной по-зицией;
- bool getFurl(TracePosition visualPos, FURL_INFO* result) – получить(если есть) свёртку по указанной начальной позиции;
- bool ensureVisible(TracePosition visualPos) – развернуть все свёртки в которых лежит данный шаг трассы;
- void saveFurls(CString fileName) – сохранить свёртки в заданный файл;
- void loadFurls(CString fileName) – загрузить свёртки из заданного файла.

Чтобы отобразить очередной шаг трассы, нужно знать начинается ли на этом шаге свёртка, и если начинается, то в каком состоянии она находится – свёрнутом или развёрнутом. Для этого используется метод `getFurl`. В зависимости от результатов запросов очередная строка может быть отображена как:

- просто текстовое представление текущей инструкции (свёртки на этом шаге нет)
- текстовое представление текущей инструкции, со значком «-» (свёртка есть, и она находится в развёрнутом состоянии)
- имя свёртки со значком «+»«-»(свёртка есть и она находится в свёрнутом состоянии)

Кроме того, нужно знать точную длину видимой трассы, с учётом свёрнутых участков. Для этого используется метод getStepsCount.

При переходе в заданный пользователем шаг нужно убедиться, что он в данный момент виден. Для этого используется метод ensureVisible.

Если пользователь выполнил двойной клик по начальной инструкции свёртки нужно изменить её состояние. Для этого используется метод expandFurl.

При загрузке трассы выполняется также и загрузка информации о свёртках – методом loadFurls.А после окончания работы с трассой, в слу-чае если состояние компонента FurlManager изменилось (появились новые свёртки или старые были удалены – метод isDirty возвращает истину) информация о свёртках сохраняется в файл с помощью метода saveFurls.

В среде TrEx имеется возможность выполнять свертки не только сугубо в рамках графического интерфейса, но и на уровне шагов трассы, когда

последовательность инструкций заменяется т.н. псевдоинструкцией – шаги объединены в один элемент, но с сохранением информации о зависимостях по данным. Таким образом, в псевдоинструкции сохраняется, какие данные являются для этого блока входными (читаются в данном блоке и участвуют в преобразованиях), а какие являются выходными (их значения меняются в данном блоке) и как они между собой связаны. В частности это требуется для ускорения работы алгоритмов слайсинга, так как позволит, проанализировав зависимости некоторого блока один раз и сохранив их, при следующем проходе слайсинга пропускать блок, повторно используя уже построенные зависимости.

При построении псевдоинструкции связи по данным внутри блока переводятся в представление графа зависимостей.

Граф зависимостей представляет собой направленный двудольный граф, где одно множество вершин – это ячейки, содержащие значения, а второе множество – инструкции (шаги трассы), которые читают или пишут значения в эти ячейки. Направление ребра определяется тем, пишет инструкция в ячейку или читает из неё. Если ячейка читается – ребро направлено от ячейки к инструкции, если пишется, то ребро направлено от инструкции к ячейке. Задача построения свёртки функций по графу зависимостей состоит в трансформации этого графа в другой направленный двудольный граф с атрибутированными рёбрами, который и назовём свёрткой. Дадим его определение.

Назовём свёрткой такой двудольный граф, в котором:

- Одно множество состоит из входных ячеек (вершин) исходного графа зависимостей, то есть таких, для которых нет входящие рёбер в графе зависимостей.
- Второе множество состоит из выходных ячеек (вершин) исходного графа зависимостей, то есть таких, для которых существуют входящие рёбра в графе зависимостей.
- Рёбра направлены строго из первого множества во второе.
- Атрибут на ребре обозначают тип зависимости между входной и выходной ячейкой, которые соединяет ребро.

Выделим несколько видов зависимостей между ячейками.

- Простая связь по данным – значение ячейки A во входной точке блока влияет на значение ячейки B в выходной точке блока. Таким зависимостям соответствуют любые инструкции перемещения значения из одной ячейки в другую (например MOVE) и преобразования ячейки (например ADD)
- Косвенная связь по данным – адрес ячейки B в выходной точке блока зависит от значения ячейки A во входной точке блока; Таким зависимостям соответствуют операции с указателями, при которых адрес ячейки вычисляется на основании значения другой ячейки, например при работе с массивом: a[i] – адрес

искомой ячейки памяти зависит от значения индекса i.

Рассмотрим подробнее преобразование «свёртка». Фактически требуется для всех входных и выходных ячеек, между которыми существует путь по рёбрам графа зависимостей добавить ребро в граф свёртки. Однако возникает проблема задания атрибута этого ребра. Таким образом, требуется:

1. Для каждой пары ячеек, одна из которых является входной, а другая выходной в графе зависимостей, найти все пути между ними.
2. Для каждого найденного пути на основе встречающихся в нём инструкций сгенерировать атрибут типа зависимости между входной и выходной ячейкой.
3. На основании же имеющегося атрибута, полученного при анализе прошлых путей и вновь созданного атрибута получить итоговый атрибут типа зависимости.

Алгоритм поиска путей в графе является стандартным, поэтому остановимся подробнее на 2 и 3 пункте. Рассмотрим различные виды путей типы атрибутов, им соответствующие.

- Путь из простых зависимостей. То есть входная переменная x участвовала в некоторых вычислениях, результатом которых явилось значение выходной переменной y. В этом случае атрибут, полученный на основании всего пути, соответствует простой зависимости y(x).
- Путь из простых зависимостей, в котором присутствует одна или несколько косвенных зависимостей. В этом случае атрибут, полученный на основании всего пути, соответствует косвенной зависимости y(x).

Из вышесказанного следует, что приоритет выставления атрибутов при анализе зависимостей следующий:

- Косвенная зависимость.
- Простая зависимость.

Рассмотрим теперь процесс «слияния» атрибутов, полученных при анализе разных путей от входной ячейки к выходной. Вначале рассмотрим пример.

Пусть a – текущая выходная ячейка блока, b – текущая входная ячейка блока, a c и d – некоторые другие выходные ячейки блока. Пусть между ячейками c и b есть простая зависимость, а между ячейками d и b – зависимость косвенная. Пусть есть инструкция, реализующая простую зависимость a(c,d). Требуется определить атрибут зависимости a(b). Видно, что существует 2 пути из b в a: b – c – a и b – d – a. Причём первому пути соответствует атрибут простой зависимости, а второму – атрибут косвенной зависимости.

В этом случае итоговый атрибут будет соответствовать «объединению» этих двух типов зависимости. Очевидно, что в случае объединения атрибутов одного типа, итоговый атрибут будет равен каждому из исходных. Следовательно, атрибуты на рёбрах могут принимать три значения:

- атрибут простой зависимости
- атрибут косвенной зависимости
- атрибут объединения простой и косвенной зависимости.

Предлагается хранить зависимости выходных ячеек от входных в виде битовых масок, где отдельные биты соответствуют типам зависимостей. А объединённым зависимостям соответствуют группы выставленных битов. Это позволит эффективно добавлять новые типы зависимостей, не меняя модели.

В случае представления значения атрибута, как битовой маски, объединение атрибутов представляет собой просто выполнение побитового "или" над их масками.

Алгоритм построения свёрток зависимостей по заданному блоку инструкций реализован в виде компонента `FurlMaker`. Промежуточные результаты алгоритма в процессе его работы хранятся в специальном компоненте, `DependencyContainerTree` оптимизированном по скорости доступа с учётом специфики алгоритма. Окончательные результаты хранятся в компоненте `CDependencyFurl`, который аналогичен компоненту визуальных свёрток, но дополнительно хранит отображение входных элементов на выходные и наоборот с учётом видов зависимостей.

Основные методы алгоритма `FurlMaker`:

- `analyzeMinstr()` – проанализировать текущую микроинструкцию;
- `removeLocals()` – убрать из зависимостей временные ячейки (используются для промежуточного хранения значений);
- `fillFurl()` – Заполнение свёртки найденными зависимостями;
- `addDependency()` – Функция добавления зависимости;

Для визуализации свёрток зависимостей была использована сторонняя библиотека Microsoft GLEE. Так как библиотека была написана на C#, то для её использования в C++ проекте был написан специальный переходник с использованием промежуточного языка Managed C++.

Класс `DependencyExtractor` осуществляет извлечение требуемых зависимостей из компонента `CDependencyFurl`. Необходимость этого класса обусловлена тем, что для больших свёрток количество зависимостей может быть велико и для улучшения восприятия пользователь может выбрать, как ячейки, зависимости между которыми его интересуют, так и виды зависимостей.

3.3. Связывание интерфейсов среды TrEx со скриптовым языком

В качестве инструментального языка, при разработке среды TrEx был использован язык C++, что позволило добиться в критических местах кода

должного уровня производительности. Однако с точки зрения пользователя среды она является «монолитом», обладающим фиксированным набором возможностей, и предоставляющим их средствами графического интерфейса. Разработка и встраивание собственных модулей-расширений способна решить эту проблему, однако такой подход сопряжен со значительными временными задержками, не приемлемыми для пользователя. Подходящим решением в данной ситуации является встраивание в среду интерпретатора скриптового языка, обеспеченного привязкой к методам и классам Common API.

В качестве кандидата на встраивание рассматривались следующие языки: Lua, Ch, Java/Javascript, Perl, Ruby. Были сформулированы следующие обязательные требования, которым должен удовлетворять скриптовый язык и его окружение.

- Возможность удобной работы с классами языка C++.
- Поддержка шаблонов.
- Перехват исключительных ситуаций.
- Наличие средств автоматизации для построения привязки.

В результате рассмотрения была выбрана связка Lua/SWIG. Ключевыми достоинствами выбранного решения является: открытый код, наличие сообщества, занимающегося развитием этих программных средств, малые накладные расходы на встраивание при использовании генератора кода привязки SWIG.

Возможность выполнения Lua-скриптов обеспечивается скриптовой консолью, совмещенной с окном вывода информационных сообщений среды. Применяемый интерпретатор Lua, по сравнению со стандартным, обладает следующими отличиями.

1. Тип `number`, соответствующий в стандартном Lua C-типу `double`, в TrEx является 64-битным знаковым целым.
2. Формат ввода чисел соответствует стандартному: число «10» воспринимается как десятичное 10, число «0x10» как десятичное 16, число «010» как восьмеричное 8. Формат вывода чисел всегда шестнадцатеричный, без префикса «0x».
3. Операция возведения в степень в силу слабой применимости для целых чисел заменена на «исключающее или».
4. Набор стандартных библиотек сохранен без изменений, за исключением библиотеки `math`, которая отключена по причине переопределения типа `number`.

При открытии трассы в ее контексте автоматически выполняется скрипт autoload.lua, расположенный в каталоге TrEx. Предоставляемый скрипт создает окружение, позволяющее получить доступ к элементам трассы в более простом виде, нежели при использовании C++-интерфейсов напрямую.

В стандартном окружении определен набор таблиц, используемых для ввода-вывода и работы с моделью и трассой.

Таблица `TrEx.Util` содержит следующие два поля.

- `TrEx.Util.loggerInstance` хранит ссылку на объект `Logger`, позволяющий выводить сообщения в консоль и файл журнала.
- `TrEx.Util.traceContext` хранит ссылку на объект `TraceContext` открытой трассы.

Таблица `TrEx.Model` используется для работы с регистровым файлом архитектуры, соответствующей открытой трассе. По ключу, соответствующему имени регистра (в нижнерегистровом или верхнерегистровом написании) или его числовому коду таблица позволяет получить описание регистра в виде структуры `NamedElement`.

Таблица `TrEx.Trace` содержит механизмы для работы с шагами трассы. Помимо получения количества шагов в трассе по ключу `n`, она позволяет обратиться к отдельному шагу по ключу, соответствующему его порядковому номеру. Помимо того, через ключ `TrEx.Trace.position` доступен номер выбранного пользователем шага.

Для доступа к декомпозиции на зависимости используется поле `Dep` объекта шага трассы, например `TrEx.Trace[0].Dep`. Данная таблица содержит количество зависимостей в ключе `n`, а также сами зависимости в виде объектов класса `Dep` по числовым ключам с 0 до n - 1.

Разбор на зависимости управляется флагами, содержащимися в `TrEx.Trace.depFlags`. В начальном состоянии флаги устанавливаются в значение 15, соответствующее наиболее полному набору флагов.

Рассмотрим пример использования `TrEx.Trace.Dep`: выводим все зависимости инструкции в шаге `stepIndex` (Рис. 2).

Таблица `TrEx.Notepad` предоставляет расширенные возможности вывода для скриптов с использованием окон «блокнотов». В каждом из таких блокнотов может содержаться список шагов и сопоставленной им текстовой информации с возможностью перехода к соответствующему шагу в трассе по двойному клику.

Для создания нового или доступа к уже существующему блокноту применяется индексированный доступ к таблице по имени блокнота (имя может являться произвольной строкой). Пример работы с блокнотом показан на Рис. 3.

```
local dumpDeps = function(stepIndex)
  local i

  -- Цикл по зависимостям.
  for i = 0, TrEx.Trace[stepIndex].Dep.n - 1 do
    -- Выводим номер зависимости и ее текстовый вид.
    print(i, TrEx.Trace[stepIndex].Dep[i])
  end
end
```

Рис. 2. Работа с зависимостями в Lua-скрипте.

```
for i = 0, TrEx.Trace.n - 1 do
  -- Проверяем eax.
  if 0 == TrEx.Trace[i].eax then
    -- Добавляем в блокнот "Zero EAX".
    TrEx.Notepad["Zero EAX"]:print(i, "На этом шаге eax
== 0")
  end

  -- Проверяем ebx.
  if 0 == TrEx.Trace[i].ebx then
    -- Добавляем в блокнот "Zero EBX".
    TrEx.Notepad["Zero EBX"]:print(i, "На этом шаге ebx
== 0")
  end

  -- Если на каком-то шаге eax и ebx равны нулю
одновременно, добавляем текст в оба блокнота
  -- и выделяем его красным (0x0000FF) цветом.
  if (0 == TrEx.Trace[i].eax) and (0 ==
TrEx.Trace[i].ebx) then
    -- Добавляем в оба блокнота красный текст.
    TrEx.Notepad["Zero EAX"]:cprint(i, "Причем ebx тоже
0!", 0x0000FF)
    TrEx.Notepad["Zero EBX"]:cprint(i, "Причем eax тоже
0!", 0x0000FF)
  end
end
```

Рис. 3. Пример использования `TrEx.Trace` и `TrEx.Notepad` для поиска шагов, где `eax == 0` или `ebx == 0`.

В блокноте, таким образом, определены два метода: `print(stepIndex, message)` и `cprint(stepIndex, message, color)`. В качестве `color` могут использоваться произвольные RGB-цвета.

4. Дальнейшие работы

На текущий момент среда TrEx предоставляет аналитику набор средств, позволяющий восстанавливать алгоритм в виде ассемблерного листинга,

нуждающегося в дальнейшей доработке. Среда может получить развитие не только за счет улучшения качества работы уже существующих инструментов, но и решения других, смежных задач, о которых говорилось ранее.

К числу таких задач следует отнести декомпиляцию в язык высокого уровня и поиск уязвимостей в бинарном коде.

Задача декомпиляции является естественным продолжением в поднятии абстракции при работе с низкоуровневым представлением программы. Ассемблерный листинг является нижней границей среди возможных статических представлений исследуемой программы. Верхней границей является представление в виде математической модели соответствующей проблемной области. Однако, даже задачу декомпиляция в традиционный язык программирования, такой как C, нельзя считать решенной. Известные декомпиляторы (Boomerang [5], DCC [6], REC [7] и Hex-Rays [8]) не всегда корректно восстанавливают структурные конструкции, языковые типы, затрудняются в распознавании библиотечных функций (за исключением Hex-Rays). Декомпиляция оптимизированной программы также нерешена – перечисленные декомпиляторы выдают результат, который либо не компилируется, либо работает некорректно, либо содержит ассемблерные вставки.

Разрабатываемый в ИСП РАН декомпилятор [9] в язык C, обладает развитыми возможностями в восстановлении типов. В ближайшее время стоит задача адаптация этого декомпилятора к среде TrEx, когда восстановление происходит не из исполняемого файла, а из внутреннего представления среды. Важной особенностью является то, что рассматриваемая программа не всегда разрабатывалась на языке C, потенциально возможно появление в трассе кода, написанного на языке ассемблера. Помимо того, исследуемая программа может содержать в себе результаты различных преобразований как с целью оптимизации, так и обфускации.

Еще одна смежная задача – выявление уязвимостей в бинарном коде. На протяжении ряда лет в ИСП РАН проводились работы, целью которых было выявление уязвимостей в исходном коде программ [10, 11]. Система выполняет поиск дефектов (ситуаций в исходном коде) в программах, написанных на Си и Си++, при помощи межпроцедурного анализа потока данных. Анализ выполняется по частям, что позволяет искать дефекты в программах неограниченного размера, а также в программах, полный набор исходного кода которых недоступен. Для обнаружения конкретных видов дефектов используются шаблоны ситуаций, в которых эти дефекты могут проявляться.

Применение соответствующих методик позволит выявлять уязвимости в бинарном коде, т.е. в тех ситуациях, когда исходный код недоступен или в рассмотрение попадает код не только самой программы, но и операционной системы, в том числе и драйверов устройств.

Основным источником данных для восстановления алгоритма является трасса, захватывающая все инструкции выполненные на процессоре. Трасса может

быть получена программной трассировкой, потактовой трассировкой на программном симуляторе, трассировкой на спец. оборудовании (например, для x86 это означает модификацию микрокодовой прошивки процессора), через гипервизор уровня аппаратной виртуализации. Последнее решение представляется наиболее перспективным, т.к. обладает рядом преимуществ перед остальными: (1) технология специально разрабатывалась для сокрытия (хоть и с другими целями) контролируемого выполнения и является стандартным решением для платформы x86, (2) потенциально низкие накладные расходы предположительно позволят не выходить за три порядка замедления, что в свою очередь даст возможность обойти распределенную антиотладочную защиту. Таким образом, трассировщик, построены на основе этой технологии будет обладать требуемым уровнем скрытности и в тоже время иметь качественно меньшие накладные расходы по сравнению с трассировщиком программного симулятора.

Перечисленные выше три задачи составят набор самодостаточных работ, значительно расширяющих возможности системы. Помимо них будут расширяться уже существующие инструменты, меняться архитектура графической части среды для выхода на достаточный уровень модульности.

Одним из направлений работ будет восстановление формата данных. Эта задача хоть и достаточно близка с некоторыми аспектами декомпиляции (восстановлением типов данных), уходит скорее в сторону восстановления протоколов, когда активно используется информация о содержимом буферов памяти, недоступная в случае статического анализа.

Однако предлагаемый в данной работе подход к анализу программ не следует рассматривать как противовес статическому анализу. Напротив, восстановление алгоритма, осуществляемое в рамках среды TrEx, следует рассматривать как «мост» ведущий к статическому анализу. Ключевым моментом в используемой методике анализа является то, что исходные данные для анализа имеют динамическую природу, в силу принципиальных технических причин, описанных в начале статьи. Однако, статическое представление программы (ее код) обладает неоспоримыми преимуществами по сравнению с динамическим представлением (трасса). К числу таких преимуществ следует отнести обобщение свойств программы, тогда как трасса представляет только один конкретный сценарий, и то, что при разработке и анализе естественным для человека представлением является статическое.

Таким образом, сформированное в рамках среды TrEx статическое представление программы должно передаваться далее соответствующей среде статического анализа. В случае, когда выходом TrEx является ассемблерный листинг, такой средой на безальтернативной основе становится уже упоминавшаяся среда Ida PRO. Работы по интеграции двух сред планируется начать уже в этом году.

Еще одним направлением развития среды TrEx будет пользовательский интерфейс, для которого ставятся задачи как количественные, так и качественные. В первом случае разработка сталкивается с трудностями, когда стандартные компоненты отображения не способны работать с тем объемом данных, что содержится в трассе. Эти задачи нельзя назвать принципиальными, но они также как и другие требуют ресурсов на свое решение. Второй тип задач требует принципиального решения как удобней и эффективней для аналитика представлять результаты работы того или иного программного инструмента.

5. Заключение

Применение среды позволяет существенно сократить временные ресурсы, требуемые на получение описания алгоритма. Рассмотрим модельный пример, в рамках которого вычисляется следующая функция (Рис. 4).

```
Function test_func(ByVal n As Integer) As Integer
      If n = 0 Then
            test_func = 0
      Else
            test_func = n + test_func(n - 1)
End Function
```

Рис. 4. Рекурсивное вычисление суммы арифметической прогрессии.

В первом случае программа – vb6 – реализована на VisualBasic версии 6.0 и скомпилирована в виде p-кода (кода виртуальной машины). Вторая программа – vbnet – реализована на VisualBasic .NET и скомпилирована в виде Managed-кода. Трасса снималась для n = 10. Ее размер составил порядка 700 МБ. В таблице 1 представлены результаты работы среды, позволившие уменьшить размер рассматриваемого кода на два-три порядка и довести его до приемлемого для аналитика значения (несколько сотен инструкций).

Таким образом, можно заключить, что создана система анализа, реализующая базовый набор программных средств, необходимых аналитику. Система успешно используется для решения ряда практических задач. Использование системы существенно (на порядки) сокращает время исследования.

	Трасса, содержащая код работы анализируемой программы в пользовательском режиме.			Инструкций в слайсе.
	Размер, МБ	Число шагов	Инструкций в листинге	
vb6	42	575 392	26 620	356
vbnet	35	484 248	62 726	143

Таблица 1. Результаты работы среды TrEx.

К числу основных возможностей среды TrEx следует отнести следующее. Разработана и реализована инфраструктура, позволяющая работать с трассой в рамках архитектурнонезависимых алгоритмов. Поддержаны архитектуры Intel64 и MIPS64. Разработаны и реализованы свертки блоков инструкций как исключительно в рамках графического интерфейса пользователя, так и с учетом зависимостей по данным. Разработан и реализован метод сигнатурного поиска библиотечных функций в трассах программ. Исследована возможность сжатия трассы программы, разработаны и реализованы соответствующие методы сжатия трассы.

Определены направления ближайших исследований для дальнейшего развития методики анализа и реализации системы. В частности, в следующее три года будут начаты работы связанные с декомпиляцией в язык высокого уровня, выявлением ошибок и слабостей в бинарном коде, трассировка средствами аппаратной виртуализации платформы Intel 64.

Литература

[1] Тихонов А.Ю., Аветисян А.И., Падарян В.А. Извлечение алгоритма из бинарного кода на основе динамического анализа. // Труды XVII общероссийской научно-технической конференции «Методы и технические средства обеспечения безопасности информации». Санкт-Петербург, 07-11 июля 2008 г. Стр. 109.

[2] AMD SimNow Simullator. http://developer.amd.com/cpu/simnow/Pages/default.aspx

[3] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. // IEEE Computer, 35(2):50–58, Feb. 2002.

[4] Wang C., Hill J., Knight J,. Davidson J. Software tamper resistance: obstructing static analysis of programs // Tech. Rep., N 12, Dep. of Comp. Sci., Univ. of Virginia, 2000.

[5] Boomerang Decompiler Home Page. http://boomerang.sourceforge.net/

[6] DCC Decompiler Home Page. http://www.itee.uq.edu.au/~cristina/dcc.html

[7] REC Decompiler Home Page. http://www.backerstreet.com/rec/

[8] Hex-Rays Decompiler SDK. http://www.hex-rays.com/

[9] K. Dolgova and A. Chernov. Automatic Type Reconstruction in Disassembled C Programs. // Proceedings of the 2008 15th Working Conference on Reverse Engineering. Pp. 202—206.

[10] В.С.Несов, О.Р.Маликов. Автоматический поиск уязвимостей в больших программах. // Известия ТРТУ. Тематический выпуск «Информационная безопасность». №7 vol. 6. Таганрог: Изд-во ТРТУ 2006. Стр. 38—44.

[11] Несов В.С., Гайсарян С.С. Автоматическое обнаружение дефектов в исходном коде программ. // Труды XVII общероссийской научно-технической конференции «Методы и технические средства обеспечения безопасности информации». Санкт-Петербург, 07-11 июля 2008. с. 107.