Комбинированный (статический и динамический) анализ бинарного кода

А.Ю.Тихонов, А.И. Аветисян
{fireboo@ispras.ru}, {arut@ispras.ru}

Аннотация. Рассматриваются проблемы анализа программы в бинарном коде для распознавания алгоритмов составляющих ее функций, выяснения особенностей реализации алгоритмов, обнаружения недокументированных возможностей. Традиционно для этих целей используются дизассемблеры и средства статического анализа потоков данных. Однако в случаях, когда составители программы приняли меры для защиты своей программы от анализа (например, использовали запакованный код, который распаковывается во время выполнения программы), статический анализ может не дать результатов. В данной статье предлагается использовать в таких случаях динамический анализ (анализ трасс программы) в дополнение к статическому. Рассматриваются проблемы, возникающие при анализе бинарных программ, и предлагаются способы автоматизации их решения. Описывается среда динамического анализа бинарного кода TrEx, разработанная и реализованная авторами статьи, и ее интеграция с известной средой статического анализа Ida Pro. Приведены примеры анализа конкретных бинарных программ.

Ключевые слова: восстановление алгоритмов по бинарному коду, поиск недокументированных возможностей в бинарном коде, методы получения трасс бинарных программ.

1. Введение

Основной задачей, в которой появляется необходимость в анализе бинарного кода, является ***задача понимания программ***, возникающая в целом ряде практических ситуаций: анализ вредоносного кода, когда исходный код отсутствует, либо недоступен; поиск недекларированных возможностей (далее – НДВ) в коммерческом ПО, которое поставляется без исходных кодов, в том числе в рамках сертификационных испытаний; поддержка унаследованных приложений, когда бинарный код приложения в результате многочисленных исправлений перестаёт соответствовать исходному коду. Целью таких исследований является получение информации об особенностях реализации алгоритмов, распознавание алгоритмов, восстановление протоколов обмена данными и форматов данных.

Для анализа бинарного используются методы статического и динамического анализа. При статическом анализе код исследуется без выполнения самой

программы, при динамическом анализе, программа исследуется на основании информации, полученной во время ее выполнения. Следует отметить, что при анализе бинарного кода могут применяться те же методы статического анализа, что и для исходного кода. Однако их применение требует предварительного получения графа вызовов функций программы и набора графов потока управления для каждой функции. Некоторые виды анализа потока данных требуют кроме того построения графа зависимостей (по данным и управлению). При различных анализах потока данных отслеживаются значения атрибутов, которые сопоставляются переменным, что приводит к необходимости наличия информации о переменных, использующихся в программе, их типах и других атрибутах.

В случае анализа исходного кода программа представлена на языке высокого уровня, что позволяет достаточно просто получить представление программы в виде набора графов: вызовы функций осуществляются по именам, передачи управления по меткам, все использующиеся переменные и их типы описаны явно. Единственным исключением является вызов функций по указателю, при котором не всегда можно сказать, какая функция (или их набор) вызывается в данной точке программы. Но и в этом случае доступен прототип вызываемой функции, что часто позволяет существенно сузить набор потенциально вызываемых функций. В бинарном коде отсутствуют переменные и типы данных в явном виде – вместо этого инструкции оперируют регистрами и ячейками памяти, отсутствуют и явные описания прототипов функций и их границ в коде.

Следует отметить, что бинарный код программы, как правило, бывает представлен в виде исполняемого файла некоторого известного формата (например, PE32 [1] для ОС Windows или ELF [2] для Linux). В данной работе предполагается, что анализируемый бинарный код представлен в виде *стандартной модели компиляции*, т.е. удовлетворяет следующим условиям:

- исполняемый файл содержит функции, глобальную область данных, кучу;
- могут использоваться виртуальные функции и динамически загружаемые библиотеки;
- поддерживается стек времени выполнения;
- каждая статическая (глобальная) переменная находится по фиксированному смещению в адресном пространстве программы;
- локальные переменные хранятся в стековом фрейме функции по фиксированным смещениям;
- фактические параметры функций передаются через стек вызывающей функцией, так что формальным параметрам функции соответствуют фиксированные смещения в её фрейме.

Кроме того, предполагается, что компоненты бинарного кода программы занимают фиксированные области памяти, и что код не является

самомодифицирующимся. Следует отметить, что этим предположениям удовлетворяет широкий класс программ, получаемых компиляцией с языков C/C++.

Исполняемый файл содержит области кода и статических данных программы, таблицы функций, импортируемых программой из сторонних, динамически загружаемых библиотек, а также список функций, экспортируемых программой, в случае, если программа сама является библиотекой. Область кода содержит последовательность функций программы, которая не является непрерывной, так как между функциями могут возникать неиспользуемые участки памяти вследствие выравнивания. Кроме того, многие компиляторы вставляют часть данных функции рядом с её кодом (статические данные). Библиотечные функции, которые при сборке программы были статически связаны с ней, содержатся в её коде и неотличимы от функций программы.

Одним из наиболее распространённых и развитых средств статического анализа бинарного кода является система интерактивного дизассемблирования IDA Pro [3], поддерживающая широкий ряд бинарных форматов и процессорных архитектур. При анализе исполняемых файлов IDA Pro позволяет проводить их автоматический анализ. При этом выполняется:

- выделение таблиц импортируемых и экспортируемых функций;
- дизассемблирование кода функций;
- выделение границ статических переменных в области данных;
- выделение параметров функций в стеке и на регистрах, а также локальных переменных функции и возвращаемого значения;
- построение графа вызовов и графа потока управления за счёт анализа адресов переходов;
- автоматическое распознавание библиотечных функций и их параметров с помощью технологии FLIRT [4].

При статическом анализе, выделение областей кода и данных, а также таблиц импорта и экспорта выполняется в процессе разбора исполняемого файла – данные о смещениях и размерах этих областей содержатся в заголовке. Для того чтобы выделить в области данных границы отдельных переменных требуется проанализировать команды обращения (чтения и записи) к этим данным из области кода. Аналогично, для того чтобы выделить границы функций требуется определить их начало, то есть адрес, на которой управление передаётся с помощью специальной инструкции вывода, и конец – адрес (точнее максимальный адрес, в случае если их несколько) специальной команды возврата. Перед анализом выполняется дизассемблирование кода, то есть представление его в виде листинга на языке ассемблера целевой архитектуры. Для дизассемблирования требуется отделять машинные инструкции от данных (дизассемблированию должны подвергаться только машинные инструкции). При этом, вследствие неразрешимости задачи разделения команд и данных на компьютерах с архитектурой Фон-Неймана,

проблемы могут возникать даже в случае файлов, удовлетворяющих стандартной модели компиляции. Кроме того, при дизассемблировании требуется восстанавливать корректную семантику операндов инструкции (задача различения констант и адресов). В общем случае эта задача также неразрешима.

Дизассемблирование начинается с точки входа в программу, которая прописана в заголовке исполняемого файла и идёт последовательно сверху вниз. Существует два основных алгоритма дизассемблирования – линейный алгоритм и алгоритм рекурсивного спуска.

Линейный алгоритм не анализирует поток управления программы, предполагая, что всё содержимое области кода является кодом. Алгоритм осуществляет линейный проход от начала до конца области кода, разбирая инструкции по порядку. При разборе очередной инструкции, определяется её размер, прибавляется к её смещению и получается адрес следующей инструкции. Преимуществом линейного алгоритма является высокая скорость дизассемблирования, основным недостатком – ошибки в дизассемблировании в случаях, когда код располагаться не непрерывно вследствие выравнивания, либо, когда в области кода содержатся статические данные программы.

Алгоритм рекурсивного спуска состоит в анализе потока управления, который начинается с точки (точек) входа в программу. При анализе инструкций, не осуществляющих передачу управления, алгоритм ведёт себя так же, как линейный. При анализе инструкций передачи управления вычисляется адрес перехода и добавляется в список адресов для дальнейшего анализа. После условной передачи управления или инструкции вызова функции анализ продолжается линейно, после безусловной передачи управления или инструкции возврата из функции последовательный анализ прерывается и осуществляется переход на следующий адрес в списке ранее добавленных, так как сразу за инструкцией может отсутствовать код (выравнивание, статические данные). Таким образом, алгоритм рекурсивного спуска позволяет автоматически определять области кода, соответствующие отдельным функциям.

Дизассемблирование в IDA осуществляется алгоритмом рекурсивного спуска, что приводит к необходимости вычислений адресов переходов и вызовов. В случае косвенной адресации вычисление адресов может оказаться невозможным, поэтому в IDA Pro встроен интерактивный отладчик, позволяющий получать значения динамически вычисляемых адресов. Однако его запуск и процесс экспорта динамической информации не автоматизирован, что усложняет применимость IDA Pro для больших программ.

Однако в некоторых случаях статический анализ может оказаться неприменимым. В качестве примера рассмотрим программу, снабженную защитой от анализа: код программы в исполняемом файле находится в зашифрованном виде и расшифровывается в процессе выполнения. В этом

случае можно попытаться снять дамп в тот момент, когда весь код будет расшифрован. Однако, в случае «ленивой» расшифровки (то есть выполняемой, только в момент передачи управления на соответствующий код) такого момента может и не существовать, если на текущих входных данных исполняется не весь код программы.

Статический анализ может быть осложнен и из-за отсутствия кода динамически загружаемых библиотек в исполняемом файле. В частности, могут отсутствовать компоненты ОС, с которыми взаимодействует программа. Это может помешать обнаружению зловредного кода, одним из способов внедрения которого является перезапись начала одной или нескольких системных функций с целью вставки инструкции перехода на зловредный код. Один из методов поиска и выделения механизмов внедрения зловредного кода выделения на основе динамического подхода описан в [5].

Рассмотренные примеры указывают на недостаточность статического подхода для анализа защищенного бинарного кода и целесообразность использования в подобных случаях методов динамического анализа.

Наиболее эффективным видится подход на основе комбинации динамического и статического анализа, при котором статическое представление дополняется данными, полученными в ходе динамического анализа. Интерактивный отладчик, в составе IDA Pro является одним из примеров реализации подобного подхода. Желательно, чтобы средство комбинированного анализа бинарного кода удовлетворяло следующим требованиям:

- возможность получения информации об адресах расположения кода и данных, а также вычисляемых адресах переходов и вызовов, необходимых для дополнения статического представления;
- полносистемный анализ, включающий код не только анализируемой программы, но и других модулей, в частности компонентов операционной системы;
- возможность анализа многопоточного ПО;
- поддержка различных процессорных архитектур.

В данной статье описывается система анализа бинарного кода, позволяющая комбинировать статические и динамические анализы, обеспечивая преодоление ограничений каждого подхода. Система является автоматизированной, и в ключевые моменты работы может потребовать от пользователя некоторых действий. В дальнейшем такого пользователя, обладающего дополнительными знаниями о работе программы, будем называть *аналитиком*. Примером ситуации, когда системе требуется помощь аналитика, является выбор функций, с которых следует начать анализ программы для его ускорения. Система позволяет аналитику решать такие задачи как поиск и выделение кода алгоритма, интересующего аналитика, восстановление входных и выходных данных найденного алгоритма,

восстановление интерфейсов между отдельными частями найденного алгоритма (в частности, параметров функций, используемых в алгоритме). Система использует промежуточное представление анализируемой бинарной программы [6], в котором программа представляется как последовательность инструкций абстрактной RISC-машины. Отметим, что в качестве такого промежуточного представления можно использовать и бит-код *LLVM* [7].

Дальнейшее изложение строится следующим образом. В разделе 2 описывается общая схема комбинированного подхода, основой которого является совместное использование статических и динамических методов, с использованием графа вызовов и графов потока управления функций, составляющих анализируемую программу (статический анализ) и бинарных трасс выполнения программы (динамический анализ) для выделения из них информации, уточняющей статическое представление. В разделе 3 описывается интерфейс взаимодействия компонент системы, реализующих статический и динамический анализ. В качестве компоненты статического анализа была выбрана среда *IDA Pro* [3], так как она реализует максимальный набор статических средств анализа. Раздел 4 содержит описание компоненты динамического анализа – среды *TrEx*, разработанной при участии авторов статьи. В разделе 5 рассматриваются примеры применения среды *TrEx* для решения различных задач обратной инженерии.

2. Общая схема комбинированного анализа

Общую схему комбинированного анализа, предлагаемую в данной статье, можно кратко описать следующим образом. Сначала с помощью среды *TrEx* выполняется динамический анализ программы в бинарном коде: генерируется трасса исполнения программы на уровне машинных инструкций, полученная на входных данных, выбранных таким образом, чтобы трасса включала выполнение исследуемой функциональности. Затем результаты исследования (граф вызовов и графы потока управления функций) передаются для уточнения системе *IDA Pro*.

Таким образом, бинарная программа анализируется в следующем порядке:

- Получение трассы выполнения программы, в которой выполняется интересующий аналитика сценарий работы программы.
- Структурирование трассы – выделение процессов и потоков, функций и их вызовов, построение графов потока управления выделенных функций по трассе.
- Поиск точки трассы, относящейся к исследуемой функциональности (выполняется аналитиком с использованием инструментальных средств среды *TrEx*).

- Выделение кода функций, выполняющих исследуемую функциональность (выполняется аналитиком с использованием инструментальных средств среды *TrEx*).
- Анализ выделенной функциональности – выделение входных и выходных данных функций, выполняющих исследуемую функциональность, восстановление внутренних интерфейсов между функциями (параметров функций) и связей между ними (зависимости по данным между параметрами).
- Восстановление структуры входных и выходных данных, а также параметров составных типов.
- Представление выделенной функциональности в виде ассемблерной программы (построение контрольного примера).
- Исследование свойств выделенной функциональности с применением статического подхода.

Генерация трассы, выполняется на специально оборудованном стенде с помощью одного из средств трассировки. Набор средств для решения этой задачи достаточно богат и включает как аппаратные (основанные, например, на интерфейсе JTAG), так и программные решения (например, на основе потактовых симуляторов AMD SimNow [8], Virtutech Simics [9]). Средства трассировки различаются по скорости работы и уровню изоляции процесса снятия трассы от анализируемой программы. Кроме того, класс исследуемой программы ограничивает набор средств, способных осуществить её трассировку: ограничения могут зависеть как от уровня защиты программы (например, программа может пытаться обнаружить факт собственной отладки или выполнения в симуляторе), так и от наличия связи программы с внешним миром по сети. В последнем случае критической становится скорость трассировки, так как при сильном замедлении сетевые соединения будут разрываться из-за превышения интервалов ожидания сетевых пакетов.

В трассе фиксируется не только последовательность выполнявшихся на процессоре инструкций, но и аппаратные прерывания, пользовательский ввод-вывод, приходящие извне сетевые пакеты. Таким образом, трасса содержит массив данных, описывающий все аспекты функционирования исследуемой программы.

Несомненным преимуществом автоматизированного динамического анализа (трассировки) является устойчивость ко многим методам защиты исполняемого кода от анализа, потому что эти методы защищают только форму представления защищаемого кода, не меняя его функциональности. Например, в случае применения для защиты программы *распаковщика*, когда исполняемый код хранится в зашифрованном виде и распаковывается только в момент исполнения, в трассе будет отражён как сам алгоритм распаковки, так и процесс работы распакованного кода [10]. Таким образом, исследование трассы позволяет автоматически получать статическое представление программы, путём проецирования трассы на адресное пространство, даже в

тех случаях, когда непосредственное извлечение и дизассемблирование кода из исполняемого файла невозможно. Кроме того, получение трассы с помощью симуляторов позволяет обойти большинство методов защиты кода от отладчиков.

После получения трассы требуется произвести первичное структурирование трассы: выделить код, относящийся к различным процессам и потокам. Для этого требуется либо фиксация трассировщиком точек переключения контекстов, либо наличие информации о ячейках памяти (регистрах), содержащих значения идентификаторов текущего процесса и потока.

На следующем шаге строится трасса более высокого уровня: последовательность инструкций, составляющих трассу, заменяется последовательностью вызовов функций. Это позволяет получить данные об областях в адресного пространства, соответствующих отдельным функциям, составляющим программу.

Представление трассы в виде последовательности вызовов функций позволяет построить *граф вызовов*: для каждого вызова определяется соответствующая функция и список точек вызова (возможно, пустой), лежащих внутри трассы рассматриваемого вызова. Если список точек вызова не пуст, для каждой новой точки вызова определяется соответствующая функция и в граф вызовов добавляется ребро от вызывающей функции, к вызываемой. Преимуществом такого графа вызовов является наличие в нём рёбер, соответствующих вычисляемым вызовам, которые невозможно получить, используя статический подход. Недостаток заключается в неполноте полученного графа – он содержит только вершины (функции), код которых исполнялся на заданных входных данных, и рёбра, соответствующие вызовам, которые попали в трассу.

После выделения границ кода функций для каждой функции строится *граф потока управления*, имеющий такие же преимущества и недостатки, что граф вызовов: неполнота, за счёт неполного покрытия кода функции трассой и большая точность за счёт возможности получения адресов переходов вычисляемых в процессе выполнения.

Если в состав анализируемой программы входит большое количество функций, то полученная структурированная трасса будет наряду с полезными данными содержать коды функций, не относящиеся к анализируемой функциональности. Для *отсеивания* из трассы ненужных данных необходимо найти в трассе точку, заведомо относящуюся к выполнению исследуемой функциональности: эта часть исследования выполняется аналитиком, который вручную отмечает в трассе искомую точку (*точку зацепления*).

Наиболее очевидным примером точки зацепления является точка вызова библиотечной функции, реализующей известную аналитику функциональность, например, функции ввода-вывода. Можно привести и другие подобные примеры. В частности, при исследовании алгоритмов шифрации часто бывают известны характерные константы, использующиеся

исследуемым алгоритмом. Если программа представлена в виде трассы, точки обращения к таким константам могут быть найдены с помощью алгоритма поиска состояния, осуществляющего последовательный просмотр состояний регистров на каждом шаге трассы.

В случае статического подхода для обнаружения таких точек используется механизм перекрёстных ссылок (IDA Pro). Для того чтобы найти вызов библиотечной функции нужно вначале идентифицировать эту функцию. В случае статического анализа для этого применяется сигнатурный анализ с использованием технологии сигнатурного распознавания функций FLIRT. Для успешного применения этого подхода необходимо пополнять базы данных сигнатур с выходом новых версий библиотек. Получение символьной информации возможно напрямую из исполняемых файлов известных форматов, путём её извлечения из таблицы экспорта, содержащейся в одном из заголовков файла.

При подключении символьной информации к трассе возникает дополнительная сложность, связанная с тем, что адреса символов указаны в виде смещения относительно начала модуля, и для вычисления абсолютных адресов требуются данные об адресах загрузки соответствующих модулей. Эти данные могут быть получены, например, из карты загрузки модулей перед началом снятия трассы или в некоторый момент в процессе снятия. При наличии символьной информации точки вызова известной функции могут быть найдены автоматически, однако в статическом представлении не сразу будут обнаружены вызовы по вычисляемым адресам.

Далее требуется определить место ввода начальных данных алгоритма и место вывода результатов его работы. Алгоритм реализуется некоторой функцией, или их набором, поэтому требуется определить эти функции и восстановить их входные и выходные параметры. В статическом подходе для этого может быть использован граф вызовов и анализ потоков данных. В случае анализа трассы для поиска функций можно воспользоваться динамическим стеком вызовов, что позволит определить ограниченный набор функций для анализа в графе вызовов. Восстановление параметров может осуществляться с использованием как статических, так и динамических методов, однако параметры составных типов, например, массивов, обычно передаются по указателю, что в случае статического подхода приводит к необходимости анализа алиасов [11]. Использование трассы упрощает эту проблему, так как позволяет точно вычислить адрес памяти, к которому обращается инструкция в заданном шаге трассы.

Знание входных и выходных данных позволяет выделить код алгоритма с помощью процедур статического слайсинга [12]. Их применение как правило сокращает объём анализируемого кода лишь незначительно, поэтому имеет смысл использовать динамический слайсинг [13]. При анализе трассы можно воспользоваться процедурой, аналогичной алгоритму слайсинга: из трассы выделяются только те инструкции, входных операндов которых достигли

входные данные, а также те инструкции, чьи выходные операнды повлияли на выходные данные алгоритма. В дальнейшем такой способ фильтрации шагов трассы будем называть слайсингом трассы. Полученный слайс программы содержит значительно меньшее количество инструкций и, как правило, является обозримым. Пример, показывающий эффективность этого метода приведён в разделе 5. Общая методика извлечения кода алгоритма на основе динамического подхода описана в [14].

Слайсинг трассы может использоваться и с целью дополнения входных и выходных данных алгоритма, в том случае, когда на начальном этапе известна только их часть. Пусть для определённости известна часть выходных данных алгоритма. Запустив обратный слайсинг на этом наборе данных с точки выхода алгоритма, в точке входа можно получить список ячеек, значение которых повлияло на заданный набор выходных данных. Этот набор ячеек добавляется к уже известным входным данным. Если при выполнении этой операции набор входных данных был расширен, требуется запустить прямой слайсинг по новому набору входных данных от точки входа в алгоритм и, аналогично, получить набор ячеек в точке выхода. Эта процедура применяется итерационно, пока не будет достигнуто состояние, когда оба множества (входных и выходных данных) перестанут расширяться.

По полученному слайсу может быть получен список функций, участвующих в обработке входных данных алгоритма и, следовательно, являющихся его частью. В случае если для некоторой их части доступна символьная информация – общую схему алгоритма можно понять, построив по трассе дерево вызовов, которое отражает последовательность вызовов функций, с учётом их вложенности.

Одним из удобных вариантов представления алгоритма является граф зависимостей, вершинами которого могут быть как отдельные инструкции – для изучения локальных частей алгоритма, так и функции (для получения глобальной схемы работы алгоритма). В первом случае, рёбра графа зависимостей отражают зависимости по данным между операндами инструкций, во втором – между входными и выходными параметрами функций. Для построения графа зависимостей на уровне функций используется уже упоминавшийся алгоритм слайсинга – для вычисления зависимостей по данным между входными и выходными параметрами одной функции, а также для получения зависимостей по данным между выходными параметрами одной функции и входными параметрами другой.

При работе с трассой важным этапом является проверка точности выделения алгоритма. Она осуществляется путём построения работоспособного ассемблерного листинга. Построение осуществляется по следующей схеме. 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.

Полученная ассемблерная программа рассматривается как контрольный пример, выполнение которого способно подтвердить корректность всех проведенных аналитиком операций, если конечно выполнение построенной программы выдает те же результаты, что и исходная программа. По требованию аналитика контрольный пример может быть декомпилирован в язык высокого уровня, либо в другое удобное для аналитика представление (например, псевдокод).

Следует отметить, что трасса позволяет частично восстановить дамп памяти, отразив в нем фактически выполнявшиеся инструкции. Этот дамп, дополненный разметкой и восстановленными символами, может быть передан подсистеме статического анализа. Такой подход обычно используется в случае, когда код в исполняемом файле был запакован (зашифрован) и, следовательно, недоступен для непосредственного извлечения. В реализованной системе в качестве компонента анализа трасс используется специально разработанная среда TrEx, а в качестве компонента статического анализа – среда IDA Pro. Их взаимодействие подробно описано в разделе 3.

Следует отметить, что описанные выше действия, выполняющиеся в процессе анализа, не зависят от типа процессорной архитектуры, к которой относится бинарный код. Это позволило реализовать единые алгоритмы анализа, избежав их дублирования для каждой архитектуры, тем самым значительно ускорив разработку, снизив трудозатраты и облегчив поддержку разработанного ПО.

3. Интеграция компонент статического и динамического анализа

Потребность в совместном использовании статического и динамического подхода обуславливается рядом причин. Основой для большинства исследований, проводимых в рамках анализа кода, таких как поиск дефектов и НДВ, является статическое представление программы, так как дефекты и НДВ обычно бывают привязаны к редко употребляемым наборам входных данных, что существенно снижает вероятность их обнаружения в ходе динамического анализа. Ошибки, соответствующие часто реализуемым путям обычно устраняются на этапе тестирования. Таким образом, основой для проведения исследований является построение полного и возможно более точного статического представления программы в виде графа вызовов и набора графов потока управления для каждой функции.

При разработке системы комбинированного анализа, в качестве компонента динамического анализа использовалась разработанная среда анализа бинарных трасс TrEx, а в качестве статического компонента – среда IDA Pro.

3.1. Цели и направления интеграции.

В рамках интеграции решались следующие основные задачи:

- дополнить статическое представление программы, информацией о динамически вычисляемых адресах, чтобы увеличить автоматически-дизассемблируемую часть программы, а также уточнить графы вызовов и потока управления, получаемые в рамках статического анализа;
- обеспечить генерацию статического представления программы по трассе выполнения, что позволит проводить анализ программ, код которых зашифрован и не может быть непосредственно извлечён из исполняемого файла;
- обеспечить возможность синхронизации между статическим и динамическим представлением программы для повышения эффективности анализа.

Для того чтобы воспользоваться возможностями среды IDA Pro требуется загрузить в неё всё статическое представление или его часть. Статическое представление программы включает в себя дамп кода и данных программы и разметку этого дампа, позволяющую, во-первых, отделить код программы от данных, а во-вторых, правильно дизассемблировать код. Дамп и разметка могут быть получены несколькими способами, применяемыми в зависимости от особенностей анализируемой программы.

3.2. Способы получения дампа.

(1) Дамп получается в результате разбора исполняемого файла или, в случае использования упаковщика, как снимок адресного пространства программы перед началом её трассировки. В обоих случаях в дамп попадёт весь код программы и её данные. Полнота информации составляет основное преимущество данного подхода. Среди ограничений возможности применения подхода отметим:

a) В ряде ассемблеров определены инструкции, позволяющие выполнять переходы по вычисляемым адресам (например, инструкция CALL mem32 для IA32). Статический анализ в этом случае не позволяет определять значения исполнительных адресов, что затрудняет автоматическое восстановление графа потока управления программы.

b) В дамп попадёт код только тех динамически загружаемых библиотек, загрузка которых будет выполнена до момента снятия дампа.

c) В случае, когда загружаемый код зашифрован (запакован), а также в случае самомодифицирующегося кода дамп не будет соответствовать исполнявшейся программе.

(2) Генерация дампа по трассе (точнее по графу потока управления, построенному по трассе), на основании анализа исполнявшегося кода. Такой механизм реализован в среде TrEx и будет подробно описан ниже. Его преимуществом является точность: дамп точно соответствует коду, который

фактически исполнялся. Это решает проблемы динамически загружаемых библиотек и защищенного кода. Однако в случае самомодифицирующегося кода для генерации однозначного дампа необходима модификация исходного кода. Ограничения применимости данного подхода:

a) Неполнота дампа – в дамп попадает только код, исполнявшийся на текущих входных данных.
b) Восстановление части дампа, содержащей значения данных в момент начала снятия трассы требует больших затрат по времени. В текущей версии среды TrEx эта часть дампа не восстанавливается.

3.3. Способы разметки дампа.

Если дамп был сгенерирован по трассе, то в качестве разметки ему можно передать адреса всех функций встречающихся в трассе – при этом IDA Pro может автоматически дизассемблировать код этих функций. В случае получения дампа из исполняемого файла или снятия снимка памяти часть кода может быть дизассемблирована в IDA автоматически, начиная от точки входа в программу и для дополнения достаточно передать адреса динамически вычисляемых переходов и вызовов. Однако в случае применения приёмов, препятствующих отладке, дизассемблирование будет неверным. Примером приёма, препятствующего отладке, может служить применение «мусорных байтов»: инструкция безусловного перехода заменяется инструкцией условного перехода с непрозрачным предикатом, который всегда выполняется; это невозможно выяснить при статическом анализе; по адресу непосредственно следующему за условным переходом размещается «мусорный байт», приводящий к ошибочному дизассемблированию последующего кода. Для преодоления таких приёмов можно передать в IDA Pro адреса начал всех исполняемых инструкций. Если при этом передавать также и коды операций, отмечая их несовпадения с данными дампа, загруженными в IDA Pro, можно автоматически выделить участки самомодифицирующегося кода.

3.4. Символьная информация.

В IDA Pro содержится достаточно богатая база данных символов, сопоставленных с адресами, для различных стандартных библиотек, кроме того есть встроенный механизм FLIRT, позволяющий «узнавать» функции на основе анализа их кода. Тем не менее, библиотеки постоянно обновляются, информация устаревает. В связи с этим реализована возможность экспорта дополнительной символьной информации из трассы в IDA Pro. Эта информация позволяет сопоставить функциям, обнаруженным в трассе их символьные имена, что значительно облегчает анализ.

3.5. Совмещение динамического и статического анализа.

При анализе программы бывает полезно одновременно наблюдать её статическое представление и её поведение на некоторых входных данных. Для реализации этой возможности в TrEx был встроен компонент синхронизации с IDA Pro: при каждом изменении шага в трассе, загруженной в TrEx, отображение в IDA Pro автоматически позиционируется на статическое представление этого шага. Таким образом, TrEx выступает в роли интерактивного отладчика, сопровождающего просмотр статического кода. Значения регистров, доступные в TrEx, а также адреса, с которых была выполнена загрузка выполнявшихся инструкций, позволяют вычислять динамически задаваемые адреса. Кроме того, совмещение статического и динамического представления удобно в случае анализа условных переходов: можно просматривать как ветвь кода, попавшую в трассу, так и альтернативную ветвь.

3.6. Реализация системы TrEx-IDA Pro.

Для обеспечения взаимодействия среды TrEx с дизассемблером IDA PRO были реализованы: модуль-расширение IDALuaPlugin, обеспечивающий доступ к интерфейсу SDK IDA Pro из Lua-скриптов; загрузчик VadsIDALoader, позволяющий загружать в IDA дамп и карту памяти в формате TrEx; клиентский модуль системы TrEx, обеспечивающий доступ к IDA Pro; генератор файла дампа и карты памяти на основе трассы.

Модуль-расширение IDALuaPlugin приспособлен для работы с версией SDK IDA PRO Advanced 5.2. Он представляет собой сервер, обеспечивающий доступ клиента к Lua-обёртке над SDK IDA Pro. В процессе запуска сервер ищет в командной строке IDA Pro адрес и порт для запуска. Для их указания нужно запустить IDA Pro с параметром: «-OIdaLua:127.0.0.1:44», подставив требуемый адрес и порт. Если параметры запуска не найдены – сервер запрашивает адрес и порт через диалог в процессе запуска. Обёртка представляет из себя (в терминах языка Lua) глобальную таблицу IdaInt функции которой соответствуют функциям SDK. Дополнительная локальная функциональность модуля-расширения – возможность вызова Lua-скриптов из файла в рамках среды IDA Pro. Для этого в меню Edit/Plugins/Lua plugin нужно указать имя Lua-скрипта. Следует отметить, что запуск модуля-расширения происходит не в момент запуска IDA Pro, а сразу после открытия выбранного файла для дизассемблирования. Старт модуля-расширения сопровождается выводом в консоль IDA Pro сообщения «IdaLua: server started <IP> : <Port>».

Загрузчик VadsIDALoader приспособлен для работы с версией SDK IDA PRO Advanced 5.2. Результат сборки проекта – динамическая библиотека VadsIDALoader.ldw. Чтобы подключить загрузчик к IDA Pro нужно положить эту библиотеку в директорию <IDA PATH>/loaders. Для использования загрузчика нужно в диалоге выбора файла для дизассемблирования указать

файл дампа, предварительно сгенерированного в среде TrEx (например memory.bin). После этого пользователю будет выдан диалог для выбора загрузчика из списка подходящих – нужно выбрать Vads dump files (вехний в списке). При этом для каждого диапазона памяти в карте памяти будут созданы соответствующие сегменты и в них загружена информация из дампа.

Внутри системы TrEx настройка взаимодействия с IDA Pro осуществляется через меню IDA Pro. Следует отметить, что только в окне базовой трассы в меню представлены все пункты – в окнах производных трасс присутствует только пункт Attach, назначение которого описано ниже.

4. Среда динамического анализа TrEx

Подробная организация среды TrEx представлена на рис. 1.

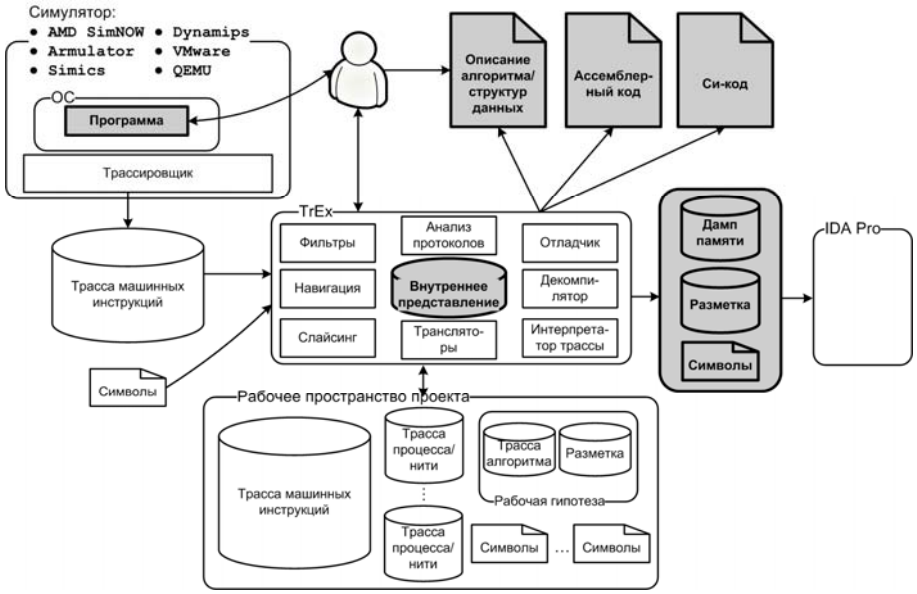

Рис. 1. Компоненты среды TrEx.

Среда предлагает следующие возможности:

- поддержка средств трассировки, обеспечивающих получение трасс для различных архитектур и способов защиты;
- возможность комбинированного анализа совместно со средой IDA Pro;
- набор генераторов различных представлений анализируемой программы;

- подсистема получения и обработки символьной информации;
- подсистема описания моделей функции; позволяет автоматически получать разметку трассы в виде комментариев к вызовам функций, содержащих значения фактических параметров вызова.

Большая часть компонент среды реализована в виде модулей расширения базовой функциональности, включающей средства работы с внутренним представлением и символьной информацией. Для обеспечения возможности встраивания подключаемых модулей TrEx в графическую среду разработан набор интерфейсных вызовов, с помощью которых модули могут указывать, каким образом следует дополнить графический интерфейс пользователя.

Центральным объектом взаимодействия между средой и модулями является интерфейс TrEx, обеспечивающий доступ к различным аспектам управления графической оболочкой. Объект этого интерфейса передаётся модулю при его инициализации. Объект TrEx позволяет получить доступ к информации об открытом проекте (трассе и подтрассах, а также вспомогательных данных) в виде объекта Workspace. Модули могут добавлять записи в журнал системы через объект Logger, который может быть получен из объектов TrEx или Workspace. Для каждой записи сохраняется дата и время её добавления, а также характер: информационная запись, предупреждение или ошибка.

Главное окно и окна трасс являются инструментируемыми, поддерживая следующие возможности: добавление (удаление) пункта меню; добавление (удаление) кнопки на панель инструментов; добавление (удаление) док-панели или панели в строке статуса. Средой производится минимальная проверка корректности работы добавлений. В частности, требуется, чтобы все добавленные в окна инструменты были удалены на момент завершения работы.

Менеджер окон трасс обеспечивает: получение упорядоченного списка всех открытых окон трасс, либо открытых окон конкретной трассы; создание нового или активацию уже открытого окна трассы; добавление плавающего окна. Кроме того, менеджер окон трасс описывает набор сигналов («окно создано», «закрытие окна», «изменение состояния окна»), на которые могут подписываться модули, чтобы иметь к ним доступ.

Для обеспечения гибкой настройки распространения изменений выбранного шага в трассе между открытыми окнами реализован следующий подход. Вся конфигурация открытых окон воспринимается в виде ориентированного графа, вершины которого соответствуют окнам, а дуги указывают на распространения изменений по выбранному шагу. Обеспечено визуальное управление описанным графом.

Все компоненты среды TrEx реализованы как задачи: по определению задача - это процедура, выполняющаяся в отдельном потоке, возвращающая по запросу другой задачи текстовое описание текущего состояния, либо значение

прогресса выполнения в виде числа от 0 до 1000, а также предоставляющая возможность прерывания пользователем с корректным освобождением ресурсов. Реализация компонентов TrEx как задач позволяет, использовать единое графическое отображение прогресса выполнения, а также вызывать одни задачи из других в качестве подзадач. Каждый вызов задачи фиксируется в журнале: сохраняется время начала и конца обработки, идентификатор и параметры задачи.

В среде реализован механизм визуальных свёрток, позволяющий при отображении пропускать части трассы, сворачивая их в древовидную структуру. Стандартный вид трассы отображает свёртки в левой части окна в виде полосок (в развёрнутом виде) или квадратов (в свёрнутом виде) различного цвета.

Одним из важных источников информации о функционировании программы является набор символов (имён) функций, которые вызываются программой. В связи с этим в среде TrEx реализована специальная подсистема получения такой информации из динамически загружаемых библиотек. Одним из важных компонентов подсистемы является разборщик библиотек наиболее распространённых форматов исполняемых файлов PE32/PE32+,COFF, ELF. Он позволяет извлекать имена экспортируемых этими библиотеками функций.

При выделении и анализе алгоритма важную роль играет знание функций и их параметров, реализующих отдельные части алгоритма. Если описание параметров некоторой функции известно, например, получено в ходе анализа или, если функция является частью известной библиотеки, то оно может быть использовано для поиска всех вызовов данной функции и получения значений фактических параметров этих вызовов. Эта информация, может быть чрезвычайно полезна для понимания работы алгоритма. В системе TrEx реализована подсистема описания моделей функций.

При создании модели функции сначала выбирается процесс, в котором должна выполняться функция, затем указываются: имя функции, начальный адрес ее кода, набор конечных адресов, связанных с функцией (может быть получен автоматически по начальному адресу), а также описания ее входных и выходных параметров. Виды параметров: параметр, передаваемый через стек с заданием смещения относительно указателя стека и размера ячейки (две константы, либо арифметических выражения); параметр-регистр (имя регистра).

5. Примеры применения среды TrEx.

Разработанный метод комбинированного анализа позволяет в полуавтоматическом режиме выделять алгоритмы, реализованные исследуемым кодом, отслеживать работу алгоритмов, распределенных по адресным пространствам нескольких процессов или использующих

особенности работы кода операционной системы. Рассмотрим несколько примеров применения комбинированного анализа.

5.1. Выделение исследуемого кода

Первый пример состоит из двух модельных программ, реализующих одну и ту же функцию вычисления суммы натурального числового ряда из N чисел.

```
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 test_func
```

Первая программа – `vb6test` – реализована на *VisualBasic* 6.0 и скомпилирована в виде p-кода (кода виртуальной машины). Вторая программа – vbnettest – реализована на *VisualBasic.NET* и скомпилирована в виде управляемого кода.

Трасса снималась для n=10. Размер трассы, полученной для каждой программы, составляет около 700Мб. Результаты обработки трассы представлены в таблице 1. После того как в трассе была обнаружена печать результатов функции `test_func`, был выполнен обратный слайсинг, в качестве критерия которого выступало результирующее значение. Время выполнения слайсинга около 50 секунд.

Из таблицы видно, что в случае ручного анализа рассматриваемых программ аналитик имел бы дело либо с миллионами шагов при динамическом анализе, либо с десятками тысяч шагов при статическом, использование слайсинга за несколько минут сводит задачу анализа к нескольким сотням инструкций, т.е. принципиально упрощает задачу аналитика.

	Трасса, содержащая код работы анализируемой программы в пользовательском режиме			Число инструкций после обработки
	Размер, МБ	Число шагов	Число инструкций в листинге	
VB v6.0	42	575 392	26 620	356
VB .NET	35	484 248	62 726	143

Таб. 1. Применение инструментов системы TrEx для сокращения размеров анализируемого кода.

5.2. Восстановление алгоритма генерации лицензионного ключа

В рамках данной задачи требовалось восстановить алгоритм генерации ключа системы SPLM (SmartPlant License Manager 2010). Трассировалась работа файла licgen.exe, который получал на вход файл machine-id.txt, содержащий идентификатор машины, а на выходе генерировал файл, содержащий лицензию в бинарном виде. Файл licgen.exe был получен в результате компиляции в отладочном режиме с прикреплённой отладочной информацией в формате pdb. В частности это привело к тому, что для всех вызовов были сгенерированы заглушки, т.е. первой инструкцией всех функций программы была инструкция JMP.

В полученной трассе были восстановлены модули и подключена символьная информация, содержащая изначально только символы системных библиотек, но не символы самой программы, т.к. pdb является закрытым форматом и напрямую получить из него символы затруднительно. В трассе был найден вызов функции WriteFile, соответствующий выводу лицензии в файл. Путём восстановления значений его параметров были восстановлены данные лицензии (предварительно были восстановлены параметры буфера, который содержал саму лицензию). Далее была получена символьная информация из файла pdb, путём загрузки исходного файла licgen.exe в дизассемблер IDA Pro, который автоматически способен извлекать символьную информацию из pdb файлов. Для генерации символьной информации в формате TextSym был применён разработан IDC-скрипт make_sym. По полученным символам была выяснена точка в трассе, где начал работать код licgen – вызов функции main. Для извлечения алгоритма генерации ключа был применён обратный слайсинг с учётом адресных зависимостей для буфера лицензии, от точки вызова функции WriteFile до точки вызова функции main. В процессе было выяснено, что запись в файл проводилась после завершения функции main, во время сброса буферов на диск, т.е. файл не был закрыт пользователем, а закрывался неявно при завершении процесса. Полученный слайс был далее отфильтрован по процессу, с удалением кода ядра. На результирующий слайс было наложено дерево вызовов функций (CallTree), что позволило эффективно представить структуру выделенного алгоритма.

5.3. Исследование зловредного кода

В рамках этой задачи требовалось исследовать поведение зловреда Trojan.Win32.Tdss.ajfl и выделить в виде ассемблерного листинга код этого зловреда, после его разворачивания в памяти зараженной машины. Виртуальная машина AMD SimNow была заражена указанным зловредом, путём запуска его исполняемого файла Trojan.Win32.Tdss.ajfl.exe. Данный зловред перехватывает любое обращение к диску на уровне файловой системы. Чтобы убедится, что система заражена был использован инспектор ядра GMER. После этого была включена трассировка и задействована команда

оболочки cmd.exe type для вывода в консоль содержимого файла Windows_default.pif. Выполнение этой команды гарантированно должно было привести к передаче управления зловреду.

В первую очередь в полученной трассе была найдена функция CreateFileW, для которой было восстановлено значение первого параметра, содержащего имя открываемого файла (_default.pif), с целью убедится, что это нужный вызов. После чего был найден вход в ядро, реализованный инструкцией SYSENTER. Затем был найден переход в модуль файловой системы ntfs.sys. Этот переход соответствовал вызову функции IofCallDriver. Первая инструкция этой функции была перезаписана инструкцией JMP, передающей управление зловредному коду, который затем осуществляет вызов одной из своих функций, используя таблицу переходов. В результате был получен один из диапазонов адресов, по которым был загружен зловредный код. По данному диапазону была проведена фильтрация трассы и построен ассемблерный листинг, который был сохранён в файл в текстовом виде.

Список литературы

[1] Tool Interface Standards (TIS). Portable Executable Formats (PE), http://www.x86.org/intel.doc/tools.htm.

[2] Tool Interface Standards (TIS). Executable and Linkable Format (ELF), http://www.x86.org/intel.doc/tools.htm

[3] IDA Pro Disassembler, http://www.hex-rays.com/idapro/.

[4] Fast Library Identification and Recognition Technology (FLIRT), http://www.idapro.ru/description/flirt/.

[5] H. Yin, Z. Liang, D. Song. HookFinder: Identifying and Understanding Malware Hooking Behaviors. // Proceeding of the 15th Annual Network and Distributed System Security Symposium (NDSS'08), Feb. 2008

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

[7] LLVM Language Reference Manual. // Llvm.org/docs/LangRef.html

[8] AMD SimNow Simulator, http://developer.amd.com/cpu/simnow/Pages/default.aspx.

[9] 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.

[10] S. Debray, J. Patel. Reverse Engineering Self-Modifying Code: Unpacker Extraction. // Proceedings of the 17th. IEEE Working Conference on Reverse Engineering, Oct. 2010, pp 131–140.

[11] 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.

[12] Weiser M. Program slicing // Proceedings of the 5th International Conference on Software Engineering. – IEEE Computer Society Press, 1981.– pp. 439–449.

[13] Korel B., Laski J. Dynamic program slicing // Information Processing Letters, Vol. 29, Issue 3.– 1988.– p. 155–163

[14] А. Тихонов, А. Аветисян, В. Падарян. Методика извлечения алгоритма из бинарного кода на основе динамического анализа. // Проблемы информационной безопасности. Компьютерные системы. 2008, №3. с. 66-71

[15] В. А. Падарян, А. И. Гетьман, М. А. Соловьев. Программная среда для динамического анализа бинарного кода. // Труды Института системного программирования, том 16, 2009, с. 51-72. –.
[16] M. Venable, M.R. Chouchane, M.E. Karim, and A. Lakhotia. Analyzing memory accesses in obfuscated x86 executables. In DIMVA'05, pages 1-18, 2005.

Combined (static and dynamic) analysis of binary code

A.YU. Tikhonov, A.I. Avetisyan
{fireboo@ispras.ru}, {arut@ispras.ru}
ISP RAS, Moscow, Russia

Abstract. This paper investigates the process of binary code analysis. To achieve typical goals (such as extracting algorithm and data formats, exploiting vulnerabilities, revealing backdoors and undocumented features) a security analyst needs to explore control and data flow, reconstruct functions and variables, identify input and output data. Traditionally for this purposes disassemblers and other static data flow analysis tools have been used. However, since developers have been taking steps to protect their programs from analysis (for example, code being unpacked or decrypted at runtime), static analysis may not yield results.

In such cases we propose to use dynamic analysis (analysis of execution traces of the program) to complement static. The problems that arise in the analysis of binary programs are discussed, and corresponding ways to automate solving them are suggested. The core of proposed method consists of whole system tracing and consecutive representation lifting: OS-aware events, process/thread identification, fully automated control and data flow reconstruction. The only manual step is searching for anchor instructions in the trace, e.g. I/O operations, which are used as input criteria for another automated step: precise algorithm extraction by trace slicing. The final step of the method constructs static test case code suitable for further analysis in tools such as IDA Pro.

We implemented the proposed approach in an environment for dynamic analysis of binary code and evaluated it against a model example and two real-world examples: a program license manager and a malware program. Our results show that approach successfully explores algorithms and extracts them from whole system traces. The required efforts and amount of time are significantly reduced as compared with traditional disassembler and interactive debugger.

Keywords: algorithm recovery, binary code, backdoors, instruction level tracing

References

[1]. Tool Interface Standards (TIS). Portable Executable Formats (PE), http://www.x86.org/intel.doc/tools.htm.
[2]. Tool Interface Standards (TIS). Executable and Linkable Format (ELF), http://www.x86.org/intel.doc/tools.htm
[3]. IDA Pro Disassembler, http://www.hex-rays.com/idapro/.
[4]. Fast Library Identification and Recognition Technology (FLIRT), http://www.idapro.ru/description/flirt/.
[5]. H. Yin, Z. Liang, D. Song. HookFinder: Identifying and Understanding Malware Hooking Behaviors. Proceeding of the 15th Annual Network and Distributed System Security Symposium (NDSS'08), Feb. 2008

[6]. V.A. Padaryan, M.A. Solov'ev, A.I. Kononov. Simulation of operational semantics of machine instructions. Programming and Computer Software, May 2011, Volume 37, Issue 3, pp 161-170, DOI 10.1134/S0361768811030030
[7]. LLVM Language Reference Manual http://llvm.org/docs/LangRef.html
[8]. AMD SimNow Simulator http://developer.amd.com/cpu/simnow/Pages/default.aspx
[9]. 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. DOI 10.1109/2.982916
[10]. S. Debray, J. Patel. Reverse Engineering Self-Modifying Code: Unpacker Extraction. Proceedings of the 17th. IEEE Working Conference on Reverse Engineering, Oct. 2010, pp 131–140.
[11]. 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.
[12]. Weiser M. Program slicing. Proceedings of the 5th International Conference on Software Engineering. IEEE Computer Society Press, 1981. pp. 439–449.
[13]. Korel B., Laski J. Dynamic program slicing. Information Processing Letters, Vol. 29, Issue 3. 1988. p. 155–163.
[14]. Tikhonov A.YU., Avetisyan A.I., Padaryan V.A., Metodika izvlecheniya algoritma iz binarnogo koda na osnove dinamicheskogo analiza [Methodology of exploring of an algorithm from binary code by dynamic analysis]. Problemy informatsionnoj bezopasnosti. Komp'yuternye sistemy. 2008, №3. pp. 66-71 (in Russian)
[15]. Padaryan V.A., Getman A.I., Solov'ev M.A. Programmnaya sreda dlya dinamicheskogo analiza binarnogo koda [Software environment for dynamic analysis of binary code]. Trudy ISP RAN [The Proceedings of ISP RAS], vol 16, 2009, pp. 51-72 (in Russian).
[16]. M. Venable, M.R. Chouchane, M.E. Karim, and A. Lakhotia. Analyzing memory accesses in obfuscated x86 executables. In DIMVA'05, pages 1-18, 2005. DOI 10.1007/11506881_1