О построении статического и динамического представлений бинарного кода по полносистемной трассе машинных инструкций*

[1] *С.С. Панасенко <spanasenko@ispras.ru>*
[1,2] *В.А. Падарян <vartan@ispras.ru>*
[1] *А.Ю. Тихонов <fireboo@ispras.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1.*

Аннотация. В статье рассматриваются два представления бинарного кода, получаемых в результате анализа полносистемных трасс. Для решения задач обратной инженерии применяются: статическое представление кода программы в виде системы с магазинной памятью и динамическое представление, описывающее потоки данных между вызовами функций, интересных для аналитика в рамках конкретной трассы выполнения. Приводится список проблем, возникающих при анализе полносистемных трасс выполнения, и предлагаются способы их решения. Описывается инфраструктура системы анализа, которая осуществляет предобработку трассы выполнения и применяется при построении представлений кода. Для статического представления разработан алгоритм построения, описаны возможные способы использования полученного представления. Для динамического представления также описан пользовательский интерфейс, позволяющий аналитику интерактивно взаимодействовать с данным представлением при решении задач обратной инженерии. В завершение статьи перечисляются направления дальнейшей работы.

Ключевые слова: бинарный код; комбинированный анализ; промежуточное представление.

1. Введение

Необходимость в анализе безопасности бинарного (исполняемого) кода возникает, когда требуется оценить фактические свойства программы: соответствие заявленным возможностям, отсутствие программных закладок, наличие дефектов и возможности их эксплуатации. Проведение анализа на уровне бинарного кода обусловлено рядом причин. Исходный код может быть

* Работа поддержана грантом РФФИ № 16-29-09632

потерян или недоступен. Характерные примеры такой ситуации – наследственные системы с частично или полностью утраченной документацией, сторонние библиотеки, вредоносное ПО. Традиционно методы анализа кода разделяют на методы статического анализа и методы динамического анализа. Статический анализ обычно менее ресурсоемок, а также позволяют исследовать различные свойства программы в целом. При этом, однако, использование статического анализа применительно к бинарному коду сталкивается со множеством затруднений. Главное из них – отсутствие статического представления в виде, на который рассчитана основная часть методов анализа. Статическое представление необходимо восстанавливать, и лишь потом открывается возможность проводить анализ. Задача восстановления представления сталкивается с проблемами, в общем случае неразрешимыми: различение кода и данных, косвенная адресация и т.п. Помимо того, набирающие популярность методы обуфскации настолько затрудняют анализ кода, что ручная работа с ним становится невозможной.

Второй подход к анализу – динамический, в свою очередь, заключается в исследовании свойств программы в процессе ее выполнения. Динамический анализ дает большие возможности в исследовании защищенного бинарного кода. Относительно легко преодолеваются такие приемы, как шифрование кода и данных, самомодификация кода программы, поскольку код преобразуется в открытый вид перед выполнением. Основной недостаток динамического анализа заключается в том, что он не дает полного покрытия кода программы. Целесообразно совместно использовать статические и динамические методы анализа, что упоминается в литературе как комбинированный анализ [1].

В публикациях ведущих зарубежных исследователей [2, 3] предлагаются подходы к исследованию безопасности исполняемого кода на основе классических компиляторных технологий, которые применяются в обратном направлении. Входные данные – исполняемый файл – покомандно транслируются в промежуточное представление, которое затем анализируется с привлечением классических компиляторных алгоритмов, таких как распространение констант, достижимые определения и др. Как и у обычных компиляторов, успех во многом определяется выразительностью промежуточного представления и удобством работы с ним в алгоритмах анализа. Известны работы, предлагающие специализированные представления кода, такие как: система с магазинной памятью в работах [4, 5], графы, представляющие одновременно потоки выполнения и потоки данных программы [6]. Применяемые в промышленных работах инструменты также формируют графовое представление изучаемого машинного кода, используя при этом традиционную форму – граф потока управления (CFG). Так, например, частичный CFG восстанавливается путем итеративного дизассемблирования исполняемого файла программы [7], после чего в полученный граф могут быть добавлены переходы по вычисляемому адресу [8], которые дизассемблер восстановить не в состоянии.

Проводимые в ИСП РАН исследования новых методов анализа бинарного кода опираются на полносистемное выполнение [9] и последующий анализ собранных трасс выполнения уровня машинных команд. Исследуемое ПО работает в виртуальной машине (ВМ), эмулирующей аппаратуру компьютера. Внутри ВМ развернута операционная система, необходимые библиотеки и сама исследуемая программа, что позволяет отслеживать поведение всей программной среды в целом. Работа с трассой представляет собой последовательное применение различных анализов, поднимающих уровень представления. На начальных этапах проводимый анализ сводится к разметке трассы, но в определенных момент накопленная информация позволяет восстановить статико-динамическое представление для наблюдаемого в трассе кода. Этот качественный переход открывает новые возможности анализа, как ручного, выполняемого человеком, так и проводимого в автоматизированном режиме.

В данной работе рассматриваются вопросы построения двух различных представлений программы. Статическое представление в виде системы с магазинной памятью (PDS) строится автоматически и используется в других видах анализа. Динамическая схема алгоритма рассчитана на интерактивное построение, когда среда анализа накопила информацию о предмете исследования. Аналитик в автоматизированном режиме управляет построением графового представления, которое наглядно демонстрирует преобразование входных данных в выходные исследуемой программой.

2. Система с магазинной памятью

Классическим представлением, используемым при статическом анализе кода, является граф потока управления (control flow graph, CFG). В качестве вершин данного графа выступают базовые блоки — непрерывные последовательности машинных команд, удовлетворяющие следующим условиям: инструкция передачи управления может являться только последней командой базового блока, а сама передача управления возможна только на первую инструкцию базового блока. При этом ребра графа потока управления соответствуют допустимым переходам между базовыми блоками.

В зависимости от решаемой задачи обычно строится либо граф потока управления отдельно для каждой анализируемой функции (процедуры), либо так называемый *межпроцедурный граф потока управления (ICFG)*, содержащий базовые блоки всего анализируемого кода со всеми возможными переходами. Использование межпроцедурного графа потока управления позволяет проводить автоматический анализ различных свойств программы, хотя такой анализ не всегда является точным, так как, во-первых, точный анализ значений, которые могут принимать переменные во время выполнения программы, является вычислительно сложной (и, в общем случае, алгоритмически неразрешимой) задачей, а во-вторых, некоторые пути в графе

потока управления оказываются *недостижимыми* (*англ.* unfeasible paths) (см. рис. 1).

```
double one, sn, cn, x;

scanf("%lf", &x);
sn = sin(x);
cn = cos(x);
double one = sn * sn + cn * cn;
if (one > 14) {
  print("Hello, world!\n");
}
```

Рис. 1: Фрагмент программы, содержащей недостижимый код.

Fig. 1: A program fragment with unreachable code.

Однако некоторое подмножество недостижимых путей можно исключить из рассмотрения и без сложного анализа значений данных программы. Важным классом таких путей являются *ложные* пути (англ. invalid paths) — пути выполнения, на которых не выполняется условие возврата из вызываемой функции в точку вызова. На рисунке 2 приведен пример такого пути.

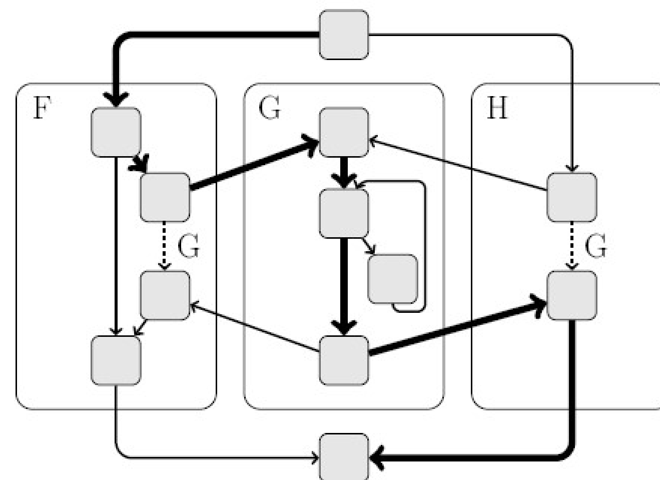


Рис. 2: Ложный путь (функции F и H вызывают функцию G).

Fig. 2: Invalid path (function G is called from functions F and H).

Одним из способов представить ICFG без ложных путей является *система с магазинной памятью* (pushdown system, PDS). Формально PDS может быть представлена в виде тройки конечных множеств $\mathcal{P} = (P, \Gamma, \Delta)$, где P — множество локаций (в качестве которых могут выступать, например, базовые блоки), Γ — множество магазинных символов, а Δ — множество правил вида

$\langle p, \gamma \rangle \to \langle p', w \rangle (p, p' \in P, \gamma \in \Gamma, w \in \Gamma^*)$. Без ограничения общности, можно считать, что $|w| \leq 2$. Этой алгебраической нотации соответствует представление CFG в виде графа, в котором базовые блоки могут быть соединены ребрами четырех типов:

1. Ребра, соединяющие базовые блоки в пределах одной функции (при переходе по таким ребрам содержимое магазина не изменяется, $\langle b_1, \gamma \rangle \to \langle b_2, \gamma \rangle$);
2. вспомогательные ребра call-to-ret, соединяющие базовый блок, завершающийся инструкцией вызова, со следующим базовым блоком данной функции (переход непосредственно по этим ребрам невозможен);
3. ребра вызова (соединяют базовый блок, завершающийся инструкцией вызова, со входным базовым блоком вызываемой функции, при переходе по такому ребру в магазин помещается идентификатор блока, в который ведет ребро call-to-ret: $\langle b_1, \gamma \rangle \to \langle b_2, b_r\gamma \rangle$));
4. ребра возврата (соединяют точку выхода функции с базовыми блоками, в которые эта функция может вернуть управление, выбор соответствующего блока осуществляется на основании содержимого магазина: $\langle b_1, b_r \rangle \to \langle b_r, \varepsilon \rangle$).

Данное представление может быть расширено для применения его в различных алгоритмах анализа потоков данных [10].

Мы представляем каждую функцию PDS в виде «гамака», входной вершиной которого является базовый блок, с которого начинается выполнение функции, а выходная вершина является фиктивным (пустым) базовым блоком — общей точкой выхода. Соответственно, для всех базовых блоков, которые возвращают управление из данной функции (напр. завершаются функцией RET), в граф добавляется ребро из этого блока в общую точку выхода. Из самого же фиктивного блока выходят только ребра возврата, передающие управление в вызывающую функцию.

На рис. 3 приводится пример PDS, содержащего код трех функций (код всех базовых блоков, кроме одного, скрыт для уменьшения размера рисунка). Пустые базовые блоки, являющиеся выходными вершинами функций, представлены на рисунке в виде эллипсов. Ребра вызова, возврата и call-to-ret выделены цветом.

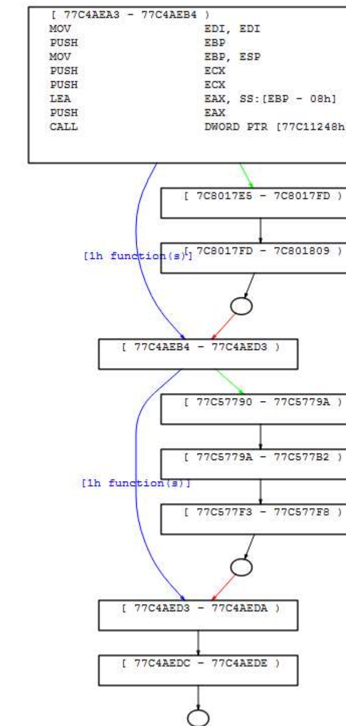


Рис. 3: Система с магазинной памятью, содержащая восстановленный код трех функций.

Fig. 3: PDS that contains code of three restored functions.

3. Проблемы при построении представления кода

Хотя сама по себе система с магазинной памятью является довольно простой структурой, построение ее по трассе машинных инструкций для реальных программ сталкивается с рядом нетривиальных трудностей.

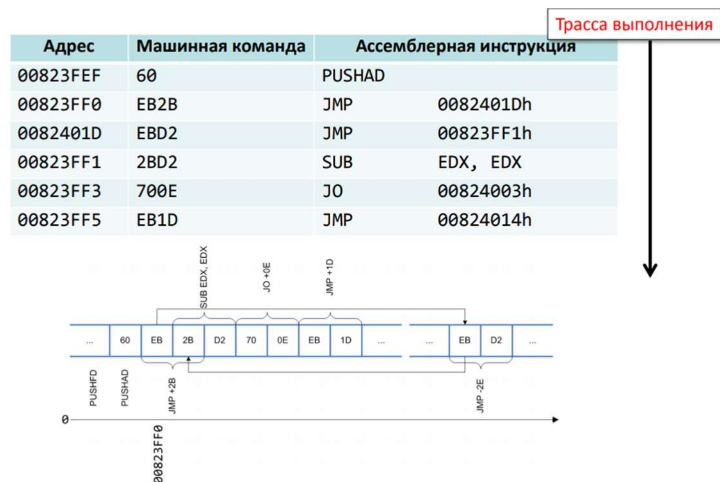
- Трасса содержит *все* инструкции, выполнявшиеся на процессоре, включая код операционной системы, обработчиков прерываний, а также других процессов, выполняемых ОС одновременно с исследуемым кодом. Для корректного построения статического представления кода необходимо:
    - распределить инструкции по процессам, к которым они принадлежат, в противном случае на одних и тех же виртуальных адресах могут быть расположены инструкции из разных исполняемых файлов;

- восстановить порядок выполнения инструкций, который может быть нарушен при одновременном выполнении нескольких нитей в рамках одного процесса.

- Трасса содержит код обработчиков прерываний. Аппаратные прерывания и исключения являются асинхронными событиями, и код обработчика прерывания может встретиться после любой инструкции. На этапе трассировки сохраняется служебная информация о том, на каких шагах трассы возникли прерывания, однако поиск точки выхода из обработчика прерывания в некоторых случаях является нетривиальной задачей. В большинстве случаев прерывания не влияют на выполнение кода программы, и после завершения обработчика прерывания выполнение программы продолжается с места возникновения прерывания, но в некоторых ситуациях это не так:
  - выполнение одной единственной инструкции может привести к возникновению нескольких прерываний (например, выполнение инструкции вызова `CALL DWORD PTR[EAX]` может вызвать несколько ошибок доступа к памяти: при попытке чтения адреса `[EAX]`, и при передаче управления вызываемой функции);
  - программные обработчики исключений могут влиять на поток управления (в частности исключения, возникшие внутри блока `try`, приводят к передаче управления на блок `catch`);
  - во время выполнения инструкции перехода может возникнуть асинхронное прерывание (такое как прерывание по таймеру), в этом случае возврат из обработчика может быть осуществлен уже на новый адрес инструкции.
- При построении статического представления кода в виде системы с магазинной памятью необходимо определить, какие из переходов являются вызовами функций либо возвратами из функций. В общем случае нельзя рассчитывать на то, что функции вызываются инструкцией `CALL` (или ее эквивалентом в другой архитектуре) и возвращают управление инструкцией `RET`. Связано это как с различными оптимизациями и особенностями целевой платформы (например, при оптимизации хвостовой рекурсии и при динамическом связывании модулей вызов функций может осуществляться инструкцией `JMP`), так и с возможным использованием методов обфускации кода (инструкция `CALL addr` может быть заменена на пару `[PUSH addr ; RET]` и т.д.).
- Кроме того, при использовании функций-заглушек, хвостовой рекурсии, динамического связывания и в некоторых других случаях

возможен одновременный возврат из нескольких функций при помощи единственной команды передачи управления. При обработке таких ситуаций необходимо корректным образом восстановить структуру PDS, добавив соответствующее число ребер возврата и ребер `call-to-ret`.

- Использование методов шифрования кода, самомодификации кода, а также динамическая загрузка модулей приводят к тому, что в рамках адресного пространства процесса одним и тем же виртуальным адресам в разное время могут соответствовать различные инструкции. При этом необходимо, с одной стороны, восстановить все базовые блоки кода, выполнявшиеся в трассе, а с другой — избежать излишнего дублирования этих базовых блоков.
- Еще одной проблемой, возникающей при построении статического представления, является перекрытие инструкций. Некоторые архитектуры вычислительных машин (например, x86) не накладывают ограничений на выравнивание команд, что позволяет реализовывать определенную технику противодействия анализу. Машинный код команд подбирается таким образом, что декодирование байт, начиная с «середины» некоторой команды приведет к выполнению осмысленного кода с требуемой семантикой. Статический дизассемблер исходит из естественного порядка выполнения команд, если управление «внутрь» команды передается прыжком назад, будет раскодирован первый вариант кода, а второй - будет успешно скрыт. В случае динамического анализа возникает проблема представления такого кода, т.к. при одинаковом содержимом памяти выполняются различные команды. На рис. 4 приведен пример такого обфусцированного кода: несколько команд перекрываются, в коде программы присутствует передача управления «внутрь» одной из инструкций. Также теоретически возможна ситуация, когда управление повторно попадает на участок памяти, содержимое которого не изменилось, но изменилось состояние процессора, отвечающее за декодирование команд.



Адрес	Машинная команда	Ассемблерная инструкция	
00823FEF	60	PUSHAD	
00823FF0	EB2B	JMP	0082401Dh
0082401D	EBD2	JMP	00823FF1h
00823FF1	2BD2	SUB	EDX, EDX
00823FF3	700E	JO	00824003h
00823FF5	EB1D	JMP	00824014h

Рис. 4: Трасса и код программы с перекрытием инструкций.

Fig. 4: Trace and binary code with overlapping instructions.

4. Используемая инфраструктура среды анализа

Алгоритм построения статического представления использует для своей работы данные, предоставляемые другими модулями среды анализа. Построению PDS предшествуют следующие этапы предобработки трассы:

1. Удаление артефактов трассировки.

Некоторые особенности используемых при трассировке инструментов могут привести появлению в снятой трассе дефектов, которые требуется исправить перед проведением дальнейшего анализа. Примером такого дефекта может служить неверное число строковых инструкций (инструкций с префиксом `REP`).

2. Разметка трассы.

Анализируемая трасса содержит как код исследуемого ПО, так и код операционной системы и других приложений, выполнявшихся внутри виртуальной машины. Для проведения анализа необходимо выделить в трассе инструкции, относящиеся к одним и тем же адресным пространствам и нитям выполнения. Для этого используется информация об операционной системе, запущенной внутри виртуальной машины.

3. Поиск диапазонов жизни не изменяющегося кода.

При анализе самомодифицирующегося (в том числе — шифрованного) кода на одних и тех же адресах памяти в разное время могут находиться разные машинные инструкции. Как при построении статического представления кода, так и при анализе трассы вручную важное значение играют участки трассы, в пределах которых код программы остается неизменным. Эти участки,

называемые в дальнейшем *поколениями*, строятся для каждого адресного пространства следующим образом:

(1) Для каждого адресного пространства создается новое поколение и пустые множества адресов `executed` и `written`.

(2) Для каждого шага трассы в множество `executed` текущего поколения добавляются адреса памяти, в которых расположена выполняющаяся на этом шаге инструкция, а в множество `written` — ячейки памяти, в которые текущая инструкция производила запись.

(3) Если на очередном шаге множества `executed` и `written` для текущего адресного пространства пересеклись, то в этом адресном пространстве создается новое поколение с пустыми множествами адресов, после чего выполнение алгоритма продолжается с шага (2).

Результатом работы данного алгоритма является множество поколений кода для каждого адресного пространства, причем можно показать, что данное множество обладает минимально возможным числом элементов.

4. Поиск обработчиков прерываний.

Асинхронные прерывания от различных устройств (таких как таймер), а также программные исключения нарушают поток выполнения программы. Инструкции, соответствующие началу кода обработчика прерывания, соответствующим образом помечаются на этапе трассировки, поиск же инструкций возврата из прерывания является нетривиальной задачей, решаемой уже в рамках среды анализа. Для решения этой задачи алгоритм поиска обработчиков прерываний пытается найти для каждого прерывания ближайший (в рамках того же потока выполнения) шаг трассы, в котором значение регистра счетчика команд (напр. `EIP`) соответствует либо значению этого же регистра перед возникновением прерывания, либо же равно адресу следующей машинной команды. Таким образом, алгоритм проверяет две гипотезы: либо инструкция, вызвавшая прерывание, не была исполнена и повторяется в трассе (напр. после подгрузки необходимой страницы памяти), либо же прерывание возникло после выполнения инструкции и изменения регистра счетчика команд.

5. Сопоставление команд вызова функций и возврата из них.

Финальным этапом предобработки трассы перед построением PDS является сопоставление команд вызова функций и возврата из них. Алгоритм, решающий эту задачу, находит в трассе команды передачи управления, которые могут использоваться для вызова функций, и помещает их в стек (при этом для каждого потока выполнения, присутствующего в трассе, используется отдельная структура данных). При обнаружении команды, соответствующей возврату из функции, алгоритм ищет парную ей команду вызова и снимает со стека необходимое количество элементов. Таким

образом, одна команда возврата управления может завершать сразу несколько вызовов, что актуально, например, в случае функций-трамплинов.

5. Алгоритм построения PDS

Алгоритм построения PDS включает в себя несколько проходов по трассе выполнения: построение базовых блоков, создание функций и проведение ребер между блоками.

5.1 Построение базовых блоков

При построении базовых блоков мы последовательно обходим все инструкции трассы, используя при этом информацию о разметке трассы и информацию об обработчиках прерываний.

Для каждого адресного пространства (процесса, присутствующего в трассе выполнения) создается набор отображений (*исполнительный адрес базовый блок*), по одному отображению на каждое поколение кода в данном адресном пространстве. В это отображение в качестве ключей добавляются адреса всех инструкций базового блока. Это позволяет, во-первых, находить в отображении базовый блок по любой его инструкции, а во-вторых, корректно обрабатывать ситуацию с перекрывающимися базовыми блоками. При этом один и тот же базовый блок может встречаться в нескольких отображениях, но не в разных адресных пространствах. Например, базовый блок `bb1` на рис. 5 состоит из трёх инструкций и трижды входит в отображения `G1` и `G2`, в качестве ключей используются адреса входящих в него инструкций.

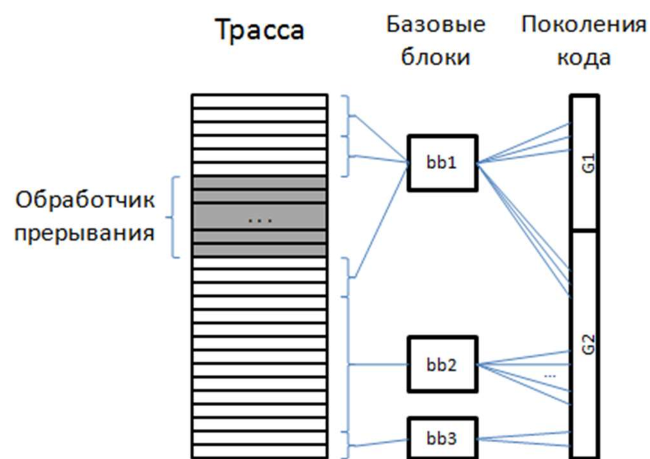


Рис. 5: Соответствие между базовыми блоками и поколениями кода.

Fig. 5: Mapping from code generations to basic blocks.

Алгоритм начинает построение базового блока с первого ещё не обработанного шага трассы.

- Сначала выполняется проверка, не принадлежит ли данная инструкция трассы некоторому известному базовому блоку. Для этого для данного шага выполняется поиск соответствующего ему адресного пространства, а в рамках этого адресного пространства — поколения кода, к которому принадлежит указанный шаг трассы. В соответствующем поколении ищется исполнительный адрес инструкции и, если он найден, алгоритм считает, что находится внутри известного блока. В этом случае алгоритм пропускает несколько шагов трассы, соответствующих инструкциям данного базового блока, и переходит к анализу следующей инструкции.
- В случае, если базовый блок в текущем поколении не найден, проверяются другие поколения кода данного адресного пространства. Среди них на том же исполнительном адресе может найтись блок, который полностью повторяется в текущем поколении. Такой блок без изменений включается в текущее поколение, после чего алгоритм также пропускает нужное число шагов трассы.
- Наконец, может оказаться, что подходящего базового блока в текущем адресном пространстве не найдено. В этом случае алгоритм создает новый базовый блок и шаг за шагом идет по трассе до тех пор, пока не встретит инструкцию передачи управления. Все найденные инструкции при этом добавляются в новый базовый блок, а сам базовый блок после построения включается в отображения для текущего поколения кода.

Необходимо отметить, что при проходе по трассе выполнения алгоритм построения PDS использует информацию об обработчиках прерываний, «перешагивая» через них. В результате базовые блоки не разрываются. Кроме того, необходимо учитывать, что базовый блок может завершиться не только инструкцией перехода (например, при построении нового блока алгоритм может наткнуться на инструкцию, принадлежащую ранее построенному блоку, на которую было передано управление).

Поскольку трасса может содержать несколько потоков выполнения, и поскольку при проходе по трассе алгоритм «перешагивает» обработчики прерываний, в результате одного прохода алгоритм построения базовых блоков обрабатывает только часть инструкций трассы. Мы храним информацию о том, какие инструкции трассы не были посещены, в множестве непересекающихся интервалов. На каждой итерации алгоритм берет из него шаг с наименьшим номером в качестве начального. Построение базовых блоков продолжается до тех пор, пока не будут обработаны все инструкции трассы.

5.2 Создание функций

На этом этапе мы используем восстановленную информацию о вызовах функций, предоставленную инфраструктурой. Для каждого вызова функции и для каждого обработчика прерываний алгоритм ищет базовый блок, с которого в трассе начинается соответствующий вызов. Для каждого их этих входных блоков создается структура данных, соответствующая функции, а также фиктивный выходной блок данной функции.

Кроме того, в рамках каждого адресного пространства мы создаем еще одну фиктивную функцию, в которую помещаются базовые блоки из начала / конца трассы, которые не удается отнести ни к какой другой функции из-за того, что соответствующий вызов / возврат не попали в трассу выполнения.

5.3 Проведение ребер

При проведении ребер мы используем информацию о вызовах, полученную в ходе предобработки трассы.

Как и на этапе построения базовых блоков, алгоритм начинает работу с первого ещё не обработанного шага трассы. Для проведения ребер используется два стека: в первом стеке хранятся указатели на текущую функцию, во втором — указатели на базовые блоки, из которых был выполнен вызов текущей функции. Изначально в первый стек помещается либо фиктивная функция (если алгоритм начинает работу с первой инструкции трассы), либо функция-обработчик прерывания, а второй стек пуст.

Дальнейшая работа алгоритма заключается в последовательном проходе по трассе и отслеживании базовых блоков, в которых он оказывается. При этом возможны три ситуации:

- При переходе от блока `b1` к `b2` в стеке вызовов, предоставляемом инфраструктурой среды анализа, появился новый вызов. В этом случае алгоритм ищет функцию, точкой входа которой является `b2`, добавляет в её в стек функций, добавляет `b1` в стек базовых блоков, а `b1` и `b2` соединяет ребром вызова.
- При переходе от блока `b1` к `b2` из стека вызовов исчез один вызов. В этом случае алгоритм соединяет `b1` с блоком – общей точкой выхода текущей функции, добавляет ребро возврата из этого блока в `b2`, а также ребро `call-to-ret` из блока, находящегося на вершине стека базовых блоков, в `b2`. После этого из стека функций и стека базовых блоков удаляется по одному элементу.
- При переходе от блока `b1` к `b2` из стека вызовов исчезло несколько вызовов (это возможно, например, при обработке функций-трамплинов). Обработка этого случая похожа на предыдущий за исключением того, что в процессе образуются ребра возврата, соединяющие общие точки выхода функций, лежащих на вершине стека, а ребра `call-to-ret` промежуточных функций также ведут в их выходные блоки.
- Стек вызовов не меняется. В этом случае `b1` и `b2` соединяются обычным ребром.

На каждом шаге `b2` добавляется в список блоков текущей (находящейся на вершине стека) функции. Обработка трассы продолжается до тех пор, пока алгоритм не достигнет ее конца или ранее обработанного участка. После этого алгоритм начинает следующую итерацию с первой необработанной инструкции.

6. Схема алгоритма

Схемой алгоритма будем называть граф, который строится на основе трассы выполнения программы и позволяет в наглядном виде представить преобразования некоторых данных, которые осуществляет исследуемая программа. В качестве таких данных могут выступать, например: чувствительные данные, полученные от пользователя, ключи шифрования, передаваемые в качестве параметров системным функциям и т.п.

Сам же граф схемы алгоритма включает в себя два типа вершин (Рис. 6), соответствующих (1) буферам, содержащим отслеживаемые данные, и (2) блокам кода, осуществляющим обработку этих данных. Граф представляет потоки данных в процессе их обработки, являясь фактически укрупненным подграфом развертки PDG по анализируемой трассе. Укрупнение заменяет отдельные команды блоками, сокращая размер графа, буферы памяти получают аннотации, в которых выражается представление человека о формате и интерпретации этих данных.

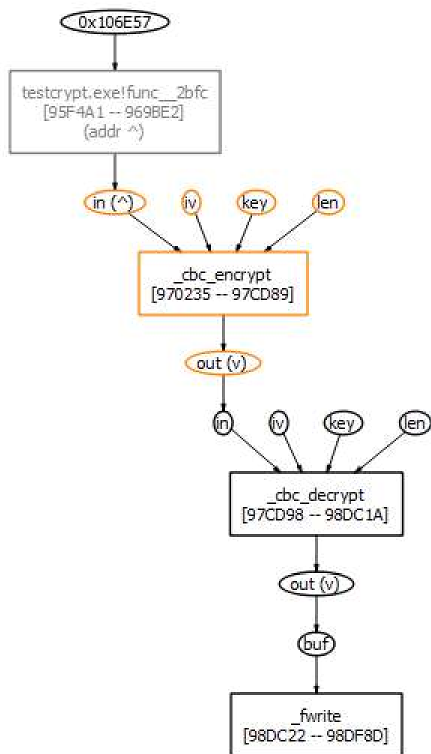


Рис. 6: Пример графа, используемого для представления схемы алгоритма.

Fig. 6: Block scheme example.

Использование схемы алгоритма может быть полезно в двух случаях: во-первых, данный граф может быть использован в качестве компонента отчета, содержащего результаты работы аналитика, а во-вторых, он может использоваться как вспомогательный инструмент при проведении анализа. Построение схемы алгоритма при этом является итеративным процессом, в ходе которого аналитик идентифицирует некоторые функции с их входными и выходными параметрами, описывает их семантику, после чего отслеживает процесс формирования (или использования) интересующих его входных (соответственно, выходных) параметров – и получает, таким образом, новое подмножество функций для анализа.

Построение схемы алгоритма опирается на значительный объем данных о программе, полученных от других алгоритмов анализа (Рис. 7). Для группировки вершин с кодом используются стек и граф вызовов; для определения, когда поток данных проходит между процессами или процессом и ядром ОС используется разметка адресных пространств; PDS используется для определения зависимостей по управлению.

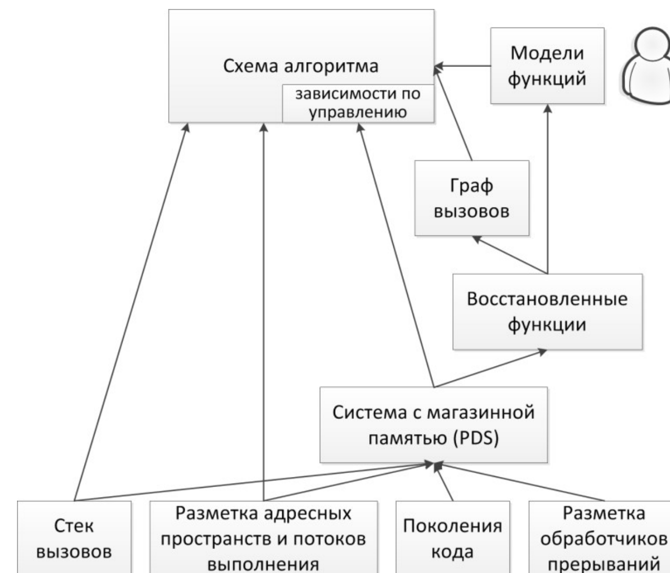


Рис. 7: Зависимость построения схемы алгоритма от других методов анализа, выполняемых над трассой машинных команд.

Fig. 7: Dependencies between analysis modules that are used to build a scheme of an algorithm.

Определенную автоматизацию в работу со схемой алгоритма вносит использование моделей функций. Модель функции — это её формальное упрощенное описание, позволяющее сгруппировать ячейки данных, с которыми взаимодействует функция, в несколько блоков данных (буферов), соответствующих известным входным и выходным параметрам функции.

Восстановление функции выражается в том, что определяются ее границы в рамках некоторого программного модуля, точка входа и точки выхода, множество именованных параметров, описывающих входные и выходные данные. Для самой функции также задается имя – оно может быть известно заранее, из каких-либо источников, либо определяется аналитиком, исходя из его понимания того, что в данной функции происходит. Параметры функции явно описывают, что с ними происходит в функции: входной, выходной, входной/выходной; для каждого параметра задается выражение, позволяющее определить значение этого параметра в некоторой точке трассы. Выражение строится над регистрами, константами, ячейками памяти. По умолчанию для входных параметров значение определяется на входе в функцию, для выходных – в момент выхода из нее. Точки определения значений параметров могут сдвигаться, исходя из особенностей ABI. Например, в архитектуре x86_64 фрейм может создаваться не сразу, некоторое время функция работает с данными в «красной зоне», после чего сдвигает верхушку стека в меньшие

адреса и обращается к данным через другое адресное выражение над регистром RSP.

Важным аспектом оказывается то, что модели функций задают зависимости между входными и выходными параметрами, которые уже получили осмысленные для человека имена и формально заданное размещение в регистрах или памяти. Итеративное построение схемы алгоритма использует подготовленную и постепенно пополняемую библиотеку моделей функций. Построение схемы предполагает управляемый человеком проход по потокам данных от некоторой заданной точки в программе до границы анализируемого алгоритма. Если поток данных в указанных границах трассы прошел по коду модели, то на схеме такой путь будет представлен единственной вершиной.

Для кода, еще не покрытого моделями, в схеме алгоритма заводятся вершины двух других типов: вызов функции, блок кода, т.е. участок кода, разделенные инструкциями вызова функций и возврата. Помимо того, в схему включаются вспомогательные вершины, соответствующие началу трассы, концу трассы, либо некоторой граничной позиции, на которой был остановлен анализ потока данных. При этом вершины данных (параметры вызова) связаны ребром с соответствующей этому вызову вершиной-моделью, а потоки данных между параметрами представлены либо ребром графа (в случае, если выходные данные одной функции с моделью непосредственно используются другой функцией с моделью), либо же подграфом-гамаком, содержащим вызовы функций без моделей. В случае если какой-либо параметр формировался (использовался) исключительно функциями, не имеющими моделей, в качестве входной (выходной) вершины гамака будет использована вспомогательная вершина.

Особенность используемого представления заключается в том, что одному и тому же фрагменту трассы (вызову функции) может соответствовать несколько вершин, расположенных в разных гамаках. Такая ситуация возникает в тех случаях, когда данные нескольких параметров совместно обрабатывались в одном и том же вызове.

7. Реализация в среде анализа

Построение рассмотренных выше представлений было реализовано в рамках среды анализа бинарного кода. Были разработаны и реализованы на языке Си++ два подключаемых модуля trawl.pds и trawl.modelgraph. Построение PDS выполняется автоматически в рамках предварительной пакетной обработки трассы, когда набор алгоритмов анализа согласованно применяются для повышения уровня представления. Модуль trawl.pds содержит ресурс, реализующий PDS и позволяющий затем воспользоваться результатами построения.

Модуль trawl.modelgraph помимо алгоритмов анализа, ресурса-схемы, используемого для хранения построенной схемы алгоритма, содержит развитый GUI, облегчающий работу пользователя. Стартовой точкой в работе

с GUI становится всплывающее окно-флоатер (Рис. 8), которое может быть вызвано из окна трассы с помощью меню `View → Open Model Graph`.

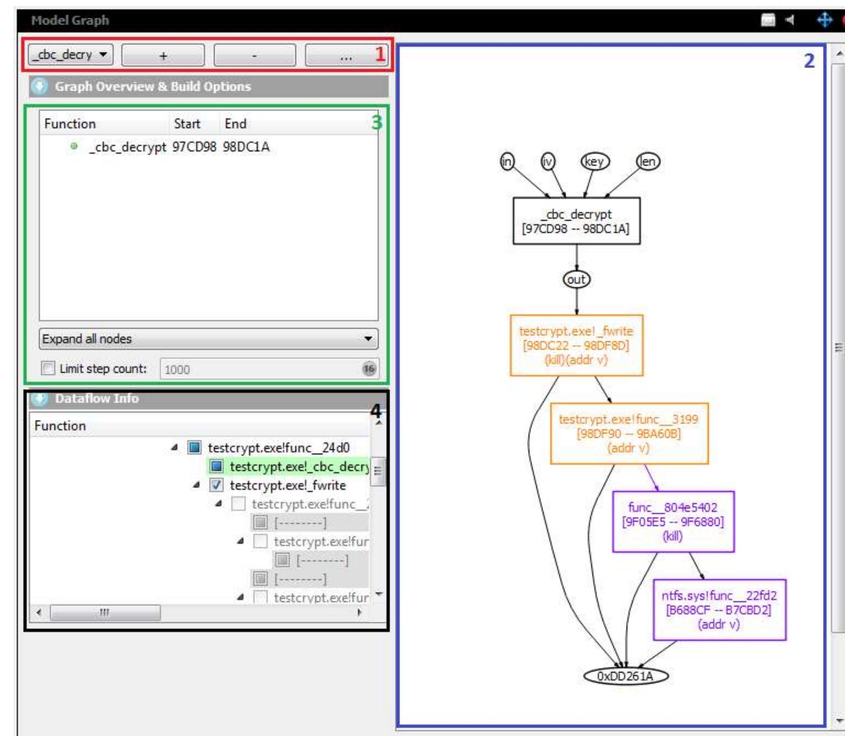


Рис. 8: Окно Model Graph.

Fig. 8: Model Graph floater.

Данное окно включает в себя несколько элементов:

- панель выбора текущего графа, позволяющую создать новый граф, либо изменить или удалить существующий;
- рабочую область, содержащую текущий граф (схему алгоритма);
- панель управления, которая содержит краткое описание текущей схемы алгоритма и позволяет настраивать параметры анализа потока данных;
- и ещё несколько элементов, которые появляются под панелью управления при различных действиях пользователя.

Ситуация, когда данные нескольких параметров совместно обрабатываются в одном и том же вызове, может представлять интерес для аналитика, соответствующие места дополнительно выделяются на графе. В настоящий момент это выделение реализовано следующим образом: при наведении

мышью на какую-либо вершину кода, на графе подсвечиваются все вершины кода, лежащие в том же диапазоне шагов, что и активная вершина.

Щелчок мышью на вершинах схемы графа приводит к выделению соответствующего вершине элемента и появлению дополнительных элементов всплывающего окна.

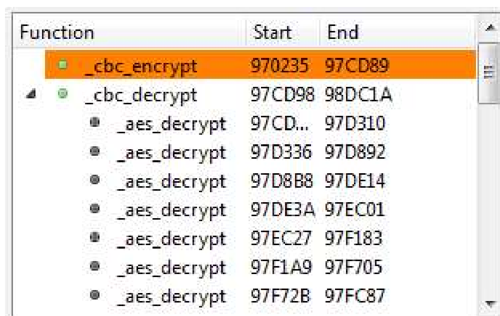
Для упрощения навигации выбранный элемент подсвечивается также в других виджетах окна, в которых он отображен. Так, например, при выборе вершины, соответствующей вызову функции с моделью, данный вызов также будет подсвечен в окне панели управления графом.

При выборе вершины, принадлежащей некоторому потоку данных, подсвечиваются все вершины этого потока данных. При этом вершины, код которых выполнялся в потоке выполнения, отличном от потока выбранной вершины, помечаются другим цветом. Так, например, при записи данных буфера out в файл на Рис. 8 выполняется как код исследуемого приложения, так и код системных библиотек, которые получают данные асинхронно.

Двойной щелчок мышью на вершинах параметров запускает анализ потока данных для данного параметра. При этом для входных параметров отслеживается обратный поток данных (т.е. процесс формирования данных этого параметра), а для выходных – прямой поток (использование данных параметра).

Вызов контекстного меню для вершины параметра позволяет выбрать направление анализа вручную.

Панель управления схемой алгоритма содержит краткое описание текущей схемы алгоритма в виде дерева вызовов функций с моделями (Рис. 9) и позволяет настроить параметры анализа потока данных.



Function	Start	End
_cbc_encrypt	970235	97CD89
_cbc_decrypt	97CD98	98DC1A
_aes_decrypt	97CD...	97D310
_aes_decrypt	97D336	97D892
_aes_decrypt	97D8B8	97DE14
_aes_decrypt	97DE3A	97EC01
_aes_decrypt	97EC27	97F183
_aes_decrypt	97F1A9	97F705
_aes_decrypt	97F72B	97FC87

Рис. 9: Краткое представление схемы алгоритма.

Fig. 9: Summary of some algorithm scheme.

При создании нового графа на панели управления отображается только «стартовая» функция, с которой и начинается анализ. В дальнейшем в дерево добавляются все функции с моделями, которые были достигнуты при анализе

потока данных в рамках работы с текущим графом. Элементы дерева могут находиться в одном из четырех состояний.

- Зеленая метка – вызов функции и его параметры показаны на графе.
- Желтая метка – на графе показаны только параметры вызова, сама же функция скрыта. Данный режим отображения полезен, если нужно исследовать (или отобразить для отчета) алгоритм, реализованный в некоторой функции, например, зависимость между ее входными и выходными параметрами.
- Белая метка – вызов и его параметры не отображаются на графе, хотя и участвовали в обработке данных.
- Черная метка – вызов функции находится внутри другого вызова, показанного на графе.

Режим отображения вызова может быть изменен с помощью контекстного меню. В настоящий момент не гарантируется связанность графа при изменении режима отображения вызовов; эта функциональность находится в разработке.

Кроме того, панель управления позволяет задать ограничения на анализ потоков данных, такие как максимальное число шагов трассы, для которых должен быть выполнен анализ, и условия остановки.

Панель управления вызовом функции с моделью появляется, если на схеме алгоритма пользователь выбрал соответствующую вершину. Панель позволяет управлять отображением параметров данного вызова и содержит краткую информацию о нем. В настоящий момент система не гарантирует связанность графа при изменении видимости параметров. Однако при скрытии и повторном отображении параметра связанность будет восстановлена.

Панель управления потоком данных появляется, если на схеме алгоритма пользователь выбрал вершину из некоторого потока данных, и позволяет управлять отображением вершин, принадлежащих текущему потоку выполнения. Панель содержит дерево, в котором листья соответствуют блокам трассы, через которые проходит выбранный поток данных, а промежуточные вершины – вызовам функций, эти блоки содержащим. Также в дереве отображаются исток и сток выбранного потока данных. С помощью кнопок-флажков, присутствующих в дереве, пользователь может объединить несколько вершин потока данных в одну. При этом свернуть вызовы, объемлющие исток и сток потока данных нельзя.

	Start	End
testcrypt.exe!func__24d0	92E5A0	9E5BCB
testcrypt.exe!func__11c0	96D85D	970223
testcrypt.exe!_aes_create_encrypt_key	96D862	96DA30
[--------]	96DA31	96DDC4
[--------]	96F2C9	96F9C0
hwi__806ecd34	96F9C1	97014F
[--------]	96F9C1	96F9EB
[--------]	970129	97014F
[--------]	970150	970223
testcrypt.exe!_cbc_decrypt	97CD98	98DC1A

Рис. 10: Дерево вызовов для выбранного потока данных.

Fig. 10: Function call tree for the selected data flow.

8. Заключение

В работе описаны представления программ, позволяющие поэтапно добиться выражения алгоритма в терминах, понятных для специалиста в прикладной области, не требуя от него специализированных технических знаний: в части архитектуры набора команд, устройства ОС и особенностей межпроцессного взаимодействия. Тем не менее, вырабатываемый в результате построения схемы граф корректно и исчерпывающее отражает все фактические стороны работы алгоритма в рамках всей вычислительной системы.

В дальнейших работах запланировано значительное развитие графического интерфейса. Предполагается снять множество ограничений по работе со схемой обусловленных тем, что представленная реализация – экспериментальный прототип.

Необходимо поддержать выделение на графе вершин, для которых в принципе есть пересечения по диапазонам шагов, дополнительно потребуется показывать, с какими именно вершинами выбранная вершина пересекается.

Планируется доработать схему алгоритма в части представления зависимостей. В настоящий момент в графе выделяются те ребра, что соответствуют передаче данных из одного потока выполнения в другой.

В планах расширить возможности свертки, включая в нее сток и исток – это может быть полезно, если поток данных содержит много фрагментов трассы, находящихся на том же уровне вложенности, что и сток/исток. Полученную вершину при этом нужно будет помечать особым образом.

Список литературы

[1]. В.А. Падарян, А.И. Гетьман, М.А. Соловьев, М.Г. Бакулин, А.И. Борзилов, В.В. Каушан, И.Н. Ледовских, Ю.В. Маркин, С.С. Панасенко. Методы и программные средства, поддерживающие комбинированный анализ бинарного кода. Труды ИСП РАН, том 26, вып. 1, 2014, стр. 251-276. DOI: 10.15514/ISPRAS-2014-26(1)-8.

[2]. Song D., Brumley D., Yin H. et al. BitBlaze: A new approach to computer security via binary analysis. Information systems security, 2008, pp. 1-25.

[3]. Brumley D., Jager I., Avgerinos T. et al. BAP: A binary analysis platform. International Conference on Computer Aided Verification, 2011, pp. 463-469.

[4]. Balakrishnan, G., Reps, T., Kidd, N., Lal, A., Lim, J., Melski, D., Gruian, R., Yong, S., Chen, C.-H., and Teitelbaum, T., Model checking x86 executables with CodeSurfer/x86 and WPDS++, (tool-demonstration paper). In Proc. Computer-Aided Verification, 2005.

[5]. Song, F., & Touili, T. (2012, August). Efficient malware detection using model-checking. In International Symposium on Formal Methods (pp. 418-433). Springer, Berlin, Heidelberg.

[6]. Caselden D., Bazhanyuk A., Payer M., McCamant S., Song D. (2013) HI-CFG: Construction by Binary Analysis and Application to Attack Polymorphism. In: Crampton J., Jajodia S., Mayes K. (eds) Computer Security – ESORICS 2013. ESORICS 2013. Lecture Notes in Computer Science, vol 8134. Springer, Berlin, Heidelberg.

[7]. Chris Eagle. IDA Pro Book, 2nd Edition / No Starch Press, 2011.

[8]. BinNavi – binary analysis IDE. Доступно по ссылке: https://www.zynamics.com/binnavi.html, 23.12.2016

[9]. Довгалюк П.М., Макаров В.А., Падарян В.А., Романеев М.С., Фурсова Н.И. Применение программных эмуляторов в задачах анализа бинарного кода. Труды Института системного программирования РАН, том 26, 2014 г. Выпуск 1. Стр. 277-296.

[10]. Reps, T., Schwoon, S., Jha, S., and Melski, D., Weighted pushdown systems and their application to interprocedural dataflow analysis. Science of Computer Programming 58, 1-2 (Oct. 2005), 206-263.

On the construction of static and dynamic representations by full-system binary code traces*

[1] *S.S. Panasenko <spanasenko@ispras.ru>*
[1,2] *V.A. Padaryan <vartan@ispras.ru>*
[1] *A.Yu. Tikhonov <fireboo@ispras.ru>*
[1] *Ivannikov Institute for System Programming, RAS,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
[2] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russian Federation

Abstract. In the paper we describe two representations of binary code obtained as a result of analysis of full-system binary traces. To solve the problems of reverse engineering we apply: a static representation of the program code in form of push-down system, and a dynamic representation that describes data flows between calls of functions, which are interesting for an analyst, within the specific execution path. We provide a list of problems that arise in analysis of the full-system traces and describe our solutions for them, including a brief description of our analysis system infrastructure, which is used for trace preprocessing. For the static representation we provide a building algorithm and list some possible ways to use it in analysis. For the dynamic representation we also describe a user interface that allows interactive interaction with user during trace analysis. At the end of the paper we describe our further work plans.

Keywords: binary code, combined analysis, intermediate representation.

References

[1]. V.A. Padaryan, A.I. Getman, M.A. Solovyev, M.G. Bakulin, A.I. Borzilov, V.V. Kaushan, I.N. Ledovskich, U.V. Markin, S.S. Panasenko. Metody i programmnye sredstva, podderzhivayushhie kombinirovannyj analiz binarnogo koda [Methods and software tools for combined binary code analysis]. Trudy ISP RAN [The Proceedings of ISP RAS], 2014, vol. 26, no. 1, pp. 251-276 (in Russian)
[2]. Song D., Brumley D., Yin H. et al. BitBlaze: A new approach to computer security via binary analysis. Information systems security, 2008, pp. 1-25.
[3]. Brumley D., Jager I., Avgerinos T. et al. BAP: A binary analysis platform. International Conference on Computer Aided Verification, 2011, pp. 463-469.
[4]. Balakrishnan, G., Reps, T., Kidd, N., Lal, A., Lim, J., Melski, D., Gruian, R., Yong, S., Chen, C.-H., and Teitelbaum, T., Model checking x86 executables with CodeSurfer/x86 and WPDS++, (tool-demonstration paper). In Proc. Computer-Aided Verification, 2005.
[5]. Song, F., & Touili, T. (2012, August). Efficient malware detection using model-checking. In International Symposium on Formal Methods (pp. 418-433). Springer, Berlin, Heidelberg.
[6]. Caselden D., Bazhanyuk A., Payer M., McCamant S., Song D. (2013) HI-CFG: Construction by Binary Analysis and Application to Attack Polymorphism. In: Crampton J., Jajodia S., Mayes K. (eds) Computer Security – ESORICS 2013. ESORICS 2013. Lecture Notes in Computer Science, vol 8134. Springer, Berlin, Heidelberg.
[7]. Chris Eagle. IDA Pro Book, 2nd Edition / No Starch Press, 2011.
[8]. BinNavi – binary analysis IDE. Доступно по ссылке: https://www.zynamics.com/binnavi.html, 23.12.2016
[9]. Dovgalyuk P.M., Makarov V.A., Padaryan, M.S. Romaneev, V.A., Fursova N.I. Primenenie programmnykh ehmulyatorov v zadachakh analiza binarnogo koda [Application of software emulators for the binary code analysis]. Trudy ISP RAN [The Proceedings of ISP RAS], 2014, vol. 26, no. 1, pp. 277-296 (in Russian)
[10]. Reps, T., Schwoon, S., Jha, S., and Melski, D., Weighted pushdown systems and their application to interprocedural dataflow analysis. Science of Computer Programming 58, 1-2 (Oct. 2005), 206-263.

* The work is supported by RFBR grant # 16-29-09632