МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМ М. В. ЛОМОНОСОВА

На правах рукописи

ГАЙВОРОНСКАЯ СВЕТЛАНА АЛЕКСАНДРОВНА

ИССЛЕДОВАНИЕ МЕТОДОВ ОБНАРУЖЕНИЯ ШЕЛЛКОДОВ В ВЫСОКОСКОРОСТНЫХ КАНАЛАХ ПЕРЕДАЧИ ДАННЫХ

Специальность 05.13.11 —
«Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей»

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель:
д. ф.-м. н., член-корр. РАН, профессор
Смелянский Р.Л.

Москва – 2014

Оглавление

Список иллюстраций

Список таблиц

Введение

С начала 2000-х годов и по сегодняшний день одним из ключевых инструментов киберпреступности являются ботнеты. *Ботнетом* называется сеть зараженных узлов, на которых запущен автономный процесс, выполняющий команды злоумышленника. Узел, на котором запущен такой процесс, принято называть *ботом* или *узлом-зомби*. Среди крупных ботнетов можно отметить Torpig, подробно исследованный командой ученых Университета Калифорнии в Санта-Барбаре [95], Zeus, по которому в 2010 году было завершено расследование ФБР и арестовано более двадцати человек [48], а так же ботнет Conficker, который привлекал внимание исследователей с 2008 года, и долгое время оставался одним из самых распространенных ботов на пользовательских компьютерах [69].

Ботнеты используются для самой разнообразной криминальной деятельности, наиболее распространенными видами которой являются:

- *Фишинг* - кража финансовой и/или приватной информации пользователей распространенного программного обеспечения. Примером такого ботнета является Storm [79].
- *Организация DDoS-атак.* DDoS-атака (Destributed Denial of Service) - распределенная атака, нацеленная на исчерпание ресурсов узла-жертвы [19]. Ботнеты, как наиболее удобный инструмент для организации подобных атак, активно используются злоумышленниками. К примеру, по статистике Arbor [1], в декабре 2013 года ежедневно были активны как минимум 1034 крупных ботнета, организующие DDoS-атаки на клиентов компании, большая часть которых является Tier-1 провайдерами. В частности, в отчете Arbor отмечены такие крупные ботнеты, как Cutwail [15], Mariposa [96], Dirt Jumper [21], Darkness [8] и другие.
- *Рассылка спама.* Спам - процесс рассылки сообщений, содержащих

коммерческий или иной контент большому числу лиц, не выражавшим желания их получать. Использование ботнетов существенно увеличивает число разосланных сообщений в единицу времени. Примером спам-ботнета является Bobax [94].

- *Кликфрод* - процесс перехода на ссылки рекламодателей или другие сайты лицом, не заинтересованном в посещении данных ресурсов. Кликфрод используется для повышения рейтинга веб-сайтов в поисковых системах, для повышения доходности рекламных площадок и др. Использование ботнета симулирует поведение большого числа легитимных пользователей [37].

Жизненный цикл любого ботнета [83] включает в себя несколько стадий (см. Рисунок 1) :

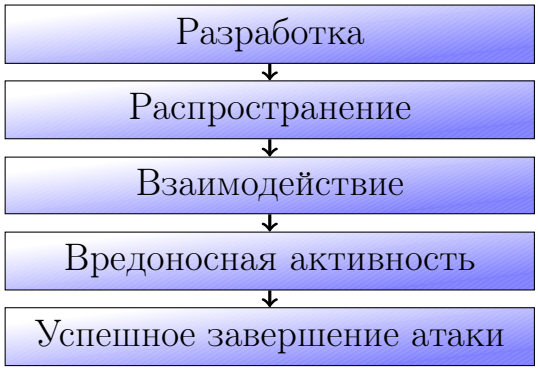


Рис. 1: Жизненный цикл ботнета.

На *стадии разработки* осуществляется планирование архитектуры ботнета и его реализация. Архитектура ботнета может быть трех типов:

1. *Централизованная* (C2C: command to control) [23], в которой боты управляются некотором выделенным узлом. Такой выделенный узел принято называть *бот-мастером*.
2. *Распределенная* (P2P: peer to peer), где каждый из ботов может выполнять роль управляющего узла [84].
3. *Гибридная*, включающая в себя несколько равноправных подсетей ботнета, каждая из которых управляется выделенным узлом.

На стадии разработки так же выбирается механизм распространения ботнета и конкретная уязвимость, которая будет эксплуатироваться при распространении.

На *стадии распространения,* в некоторой литературе так же обозначающейся как стадии заражения, осуществляется эксплуатирование уязвимостей узлов и внедрения на них ботов. За счет этого достигается подключение к инфраструктуре ботнета как можно большего количества узлов, необходимых злоумышленнику. В целях увеличения эффективности распространения и заражения большего числа узлов, на которых могут быть запущены различное программное обеспечение, ботнеты для своего распространения используют одновременно до нескольких эксплойтов - вредоносных программ, эксплуатирующих уязвимости. Часто ботнеты используют для своего распространения zero-days эксплойты - вредоносные программы, эксплуатирующие еще неопубликованные ошибки в распространенном программном обеспечении. В качестве примера можно привести ботнет, распространяющийся с помощью червя Stuxnet [31], эксплуатирующий одновременно несколько уязвимостей системы SCADA, установленной на ядерных электростанциях, и ботнет Agobot [25], использующий для своего распространения более 10 эксплойтов одновременно.

Стадия взаимодействия характеризует взаимодействие между ботами и их мастером (в случае его наличия в архитектуре ботнета). Взаимодействие между узлами ботнета разделяют на процесс регистрации - процесса, во время которого скомпрометированный узел становится эффективной частью ботнета, и процесс получения команд от мастера или группы узлов в случае распределенной архитектуры ботнета.

На *стадии активности* ботнет осуществляет вредоносную активность непосредственно.

В работе [83] рассматривается утверждение, что разрыв представленной цепочки на любой из стадий позволяет избежать потери от вредоносной

активности ботнета. Стоит заметить, что разрыв цепочки на этапе разработки ботнета требует сотрудничества с его создателями, что на практике редко достижимо. Обнаружение и фильтрация ботнета на других стадиях его жизненного цикла возможна.

В настоящей работе рассматривается проблема обнаружения и фильтрации ботнетов на этапе их *распространения*, эффективная для борьбы с ботнетами в виду своей превентивности. В тот момент, когда осуществляется стадия вредоносной активности ботнета, ущерб атакуемым узлам уже нанесен либо полностью, либо частично. В этом случае задача обнаружения и фильтрации ботнетов сводится к борьбе с последствиями вредоносной активности. Превентивное обнаружение позволит минимизировать наносимый ботнетом ущерб.

Рассматривается проблема обнаружения и фильтрации ботнетов, распространяющихся посредством удаленного эксплуатирования уязвимостей работы с памятью. *Уязвимости работы с памятью* возникают тогда, когда некоторый код в программе записывает в память больше данных, чем было предусмотрено разработчиком приложения. Типичными примерами таких уязвимостей являются переполнение стека [75], переполнение кучи [22], [38], [62], а так же некоторых других служебных структур [106], [57], [27], [89], [55]. Несмотря на то, что набирает обороты использование веб-уязвимостей для распространения ботнетов посредством drive-by-download [36], [43], [82], заражения легитимных сайтов [71], значимость удаленно эксплуатируемых уязвимостей в распространенном программном обеспечении не снижается, и вряд ли будет снижаться в ближайшее время. Большая установочная база уязвимой версии программного обеспечения обеспечивает возможность быстрого захвата значительного числа узлов. В качестве примера можно привести недавно исправленные уязвимости протокола RDP компании Майкрософт [97], уязвимости Java [91]. За последние три года, согласно статистике CVE [39], было опубликовано около 15000 удаленно эксплуатируемых

уязвимостей.

Вредоносный код, эксплуатирующий уязвимости работы с памятью, традиционно называется *шеллкодом* (shellcode) [75]. Название такого вредоносного кода обусловлено первыми атаками, эксплуатирующими уязвимости работы с памятью. Как правило, целью таких являлось получение доступа злоумышленника к консоли (shell) атакуемой системы с правами администратора. Несмотря на то, что в настоящее время с помощью таких атак возможно выполнение различной вредоносной активности, их название сохранилось.

В качестве примера рассмотрим подробнее шеллкоды, эксплуатирующие переполнения стека - один из наиболее популярных и хорошо изученных методов эксплуатирования ошибок работы с памятью. Впервые описание этой атаки было опубликовано в 1996 году в работе [75]. Тем не менее, об этой уязвимости было известно задолго до данной публикации - упоминания об этой уязвимости встречаются на протяжении как минимум 25 лет [67].

Переполнение стека возможно из-за отсутствия неявных проверок границ записываемых данных в языках C и C++ : в данных языках возможна запись большего числа данных в буфер, чем его размер. Рассмотрим эксплуатирование этой уязвимости на примере. Пусть в программе содержится некоторая уязвимая функция, в которой определяется локальный массив некоторого размера. При вызове функции в стек заносится значение *адреса возврата* - адреса, на который необходимо передать управление при завершении функции (например, на адрес инструкции, расположенной в памяти за вызываемой функцией непосредственно). Так же в стек заносится значение флага `BP`, после чего выделяется память под локальные переменные функции. Пример стека при вызове функции изображен на рисунке 2. Цель атакующего в данном случае - записать в область, отведенную под локальные переменные, специально сформированную строку бóльшей длины таким образом, чтобы адрес возврата из функции был перезаписан на

специально сформированное значение. Строка, перезаписывающая значения стека, содержит в себе вредоносные инструкции. Таким образом, цель атакующего - перезаписать адрес возврата таким образом, чтобы управление при выходе из функций передавалось на только что внедренные данные. Пример стека с внедренным шеллкодом приведен на рисунке 3.

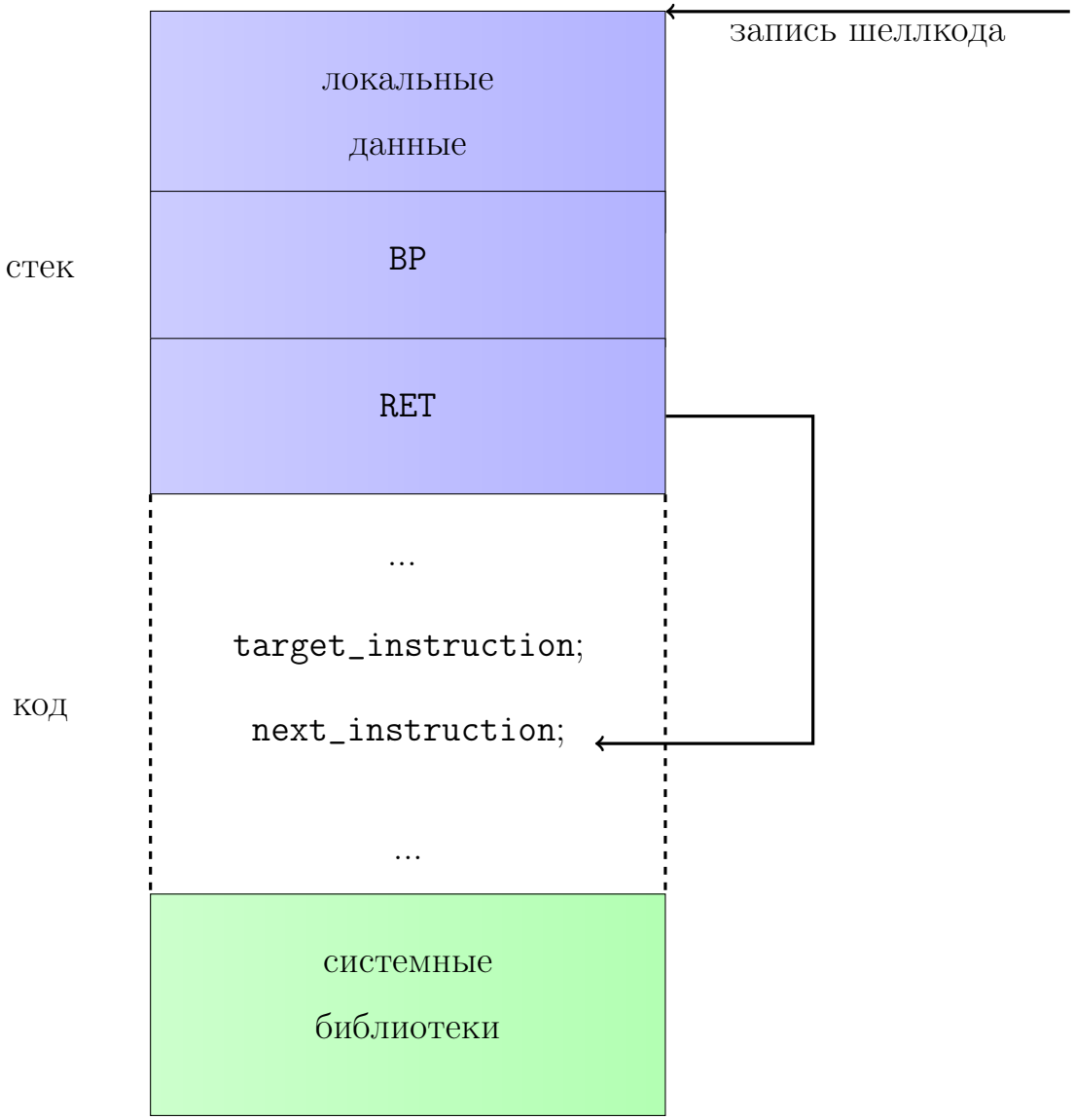


Рис. 2: Состояние стека при вызове функции `target_instruction`. Указатель `RET` содержит адрес начала следующей инструкции.

В отличие от вирусов, имеющих неограниченное число модификаций, шеллкоды имеют типичную структуру. Структуры шеллкодов, эксплуатирующих уязвимости переполнения стека, кучи, строковых переменных схо-

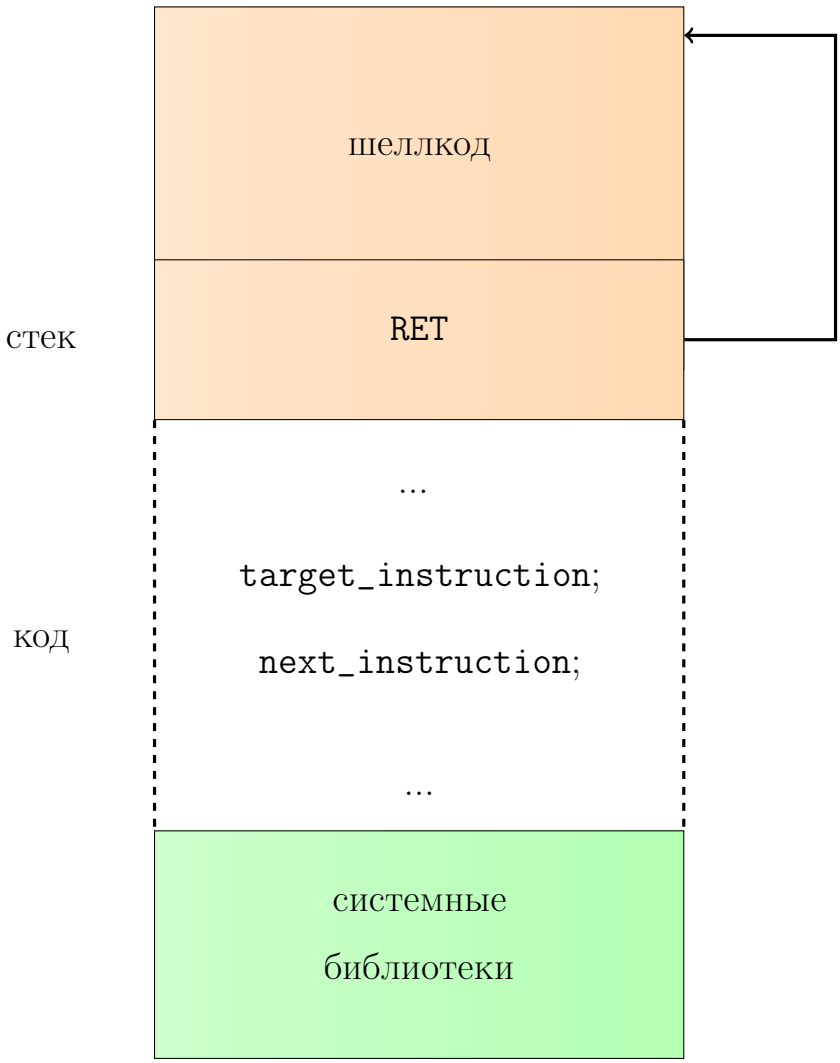


Рис. 3: Состояние стека после записи шеллкода. Указатель `RET` содержит адрес шеллкода (не обязательно начала).

жи, но могут различаться в отдельных компонентах (например, шеллкод, эксплуатирующий уязвимость переполнения кучи, не обязательно должен содержать активатор). Рассмотрим типичную структуру шеллкода, эксплуатирующего уязвимость переполнения стека (см. Рисунок 4).

Активатор представляет из себя участок тела шеллкода, гарантирующий корректное выполнение полезной нагрузки шеллкода. Под корректным выполнением здесь будем понимать выполнение функциональности, заложенной автором шеллкода. Необходимость в активаторе возникает в виду того, что зачастую атакующий не знает конкретных характеристик

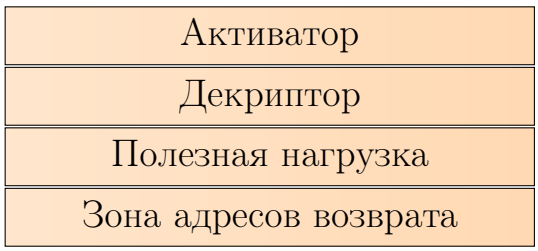


Рис. 4: Типичная структура шеллкода.

атакуемого узла: адресации, адреса начала стека и других характеристик, влияющих на выполнение внедренного кода. В случае, если управление передано на некоторое смещение от начала шеллкода, возможно выполнение совершенно иной цепочки инструкций или даже некорректной инструкции, что может привести к аварийному завершению исполнимого процесса. Активатор может быть представлен одной из следующих сущностей:

- *NOP-след* (No OPeration) - набор инструкций, обладающий следующими двумя свойствами:
  1. *NOP-след не влияет на выполнение программы.* NOP-след может быть представлен как инструкцией `nop` (`0x90`) непосредственно, так и состоять из многобайтовых инструкций, производящих в том числе и арифметические операции, но не задействующие ресурсы (регистры, память), используемые в коде далее. Таким образом, единственный эффект выполнения NOP-следа - увеличение программного счетчика.
  2. *NOP-след представляет из себя корректную цепочку инструкций целевого процессора, начиная с любого смещения.* Данное свойство гарантирует достижимость вредоносной нагрузки шеллкода, не зависимо от того, на какое смещение от начала шеллкода было передано управление (учитывая, что смещение не превышает размер NOP-следа).
- *GetPC код* (Get Program Counter) - код, вычисляющий свое расположение в адресном пространстве исполнимого процесса. GetPC код мо-

жет использоваться только с шеллкодами, содержащими декриптор. Подобный код встречается в шеллкодах в виду того, что злоумышленник не имеет возможность заранее предсказать абсолютный адрес в адресном пространстве исполнимого процесса, в который внедряется шеллкод. Тем не менее, для успешной расшифровки зашифрованной полезной нагрузки, декриптору эта информация необходима.

Декриптор. Часть тела шеллкода, необходимая для расшифровки зашифрованной части шеллкода, в случае ее присутствия. Декрипторы используются в шеллкодах, содержащих обфускации, которые активно используются злоумышленниками для уклонения от обнаружения: самораспаковка, самомодификация [105]. При этом обфусцированная часть шеллкода, производящая вредоносную активность непосредственно, до расшифровки представляет из себя набор случайных данных, а потому может быть принята различными средствами обнаружения за легитимные данные. Декриптор может представлять из себя как функцию, производящую простейшие модификации над зашифрованным телом шеллкода (`xor, and, not`), так и реализовывать более сложные криптоалгоритмы [50]. Шеллкод, не содержащий декрипторную часть, принято называть *простым шеллкодом* (plain shellcode).

Полезная нагрузка. Полезная нагрузка является неотъемлемой частью любого шеллкода. Это участок кода, выполняющий вредоносную активность непосредственно. В случае обфусцированных шеллкодов, полезная нагрузка до работы декриптора может представлять из себя последовательность случайных данных.

Зона адресов возврата. Зона адресов возврата содержит предполагаемый злоумышленником абсолютный адрес, который должен указывать внутрь NOP-следа или передавать управление на GetPC-код. Для того, чтобы увеличить вероятность перезаписывания зоны адресов возврата стека на нужное значение, оно дублируется после полезной нагрузки шеллкода

несколько раз [99].

В данной работе рассматривается задача обнаружения и фильтрации ботнетов, распространяющихся посредством шеллкодов, на высокоскоростных каналах передачи данных (например, с пропускной способностью в 1Гб/сек). Возможный пример применения решения поставленной проблемы - монитор в рамках IDS/IPS - изображен на рисунке 5. На данном рисунке каждый пакет, проходящий по каналу, анализируется на предмет содержания в нем шеллкодов. Если шеллкод в пакете обнаружен, то пакет сбрасывается.

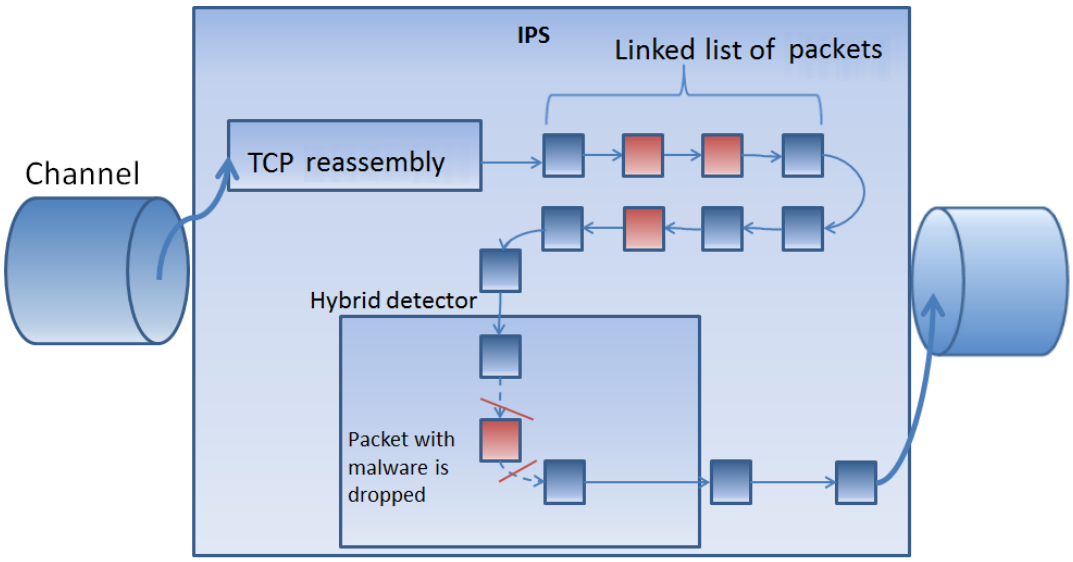


Рис. 5: Пример использования средства обнаружения и фильтрации шеллкодов.

Это, в свою очередь, накладывает определенные ограничения на решение рассматриваемой проблемы. В виду того, что высокоскоростные каналы характеризуются большим объемом передаваемой информации, проходящей по каналу в единицу времени, алгоритмическая сложность решения крайне критична. В случае высокой алгоритмической сложности возможно два сценария. При первом проходящие по каналу данные будут обработаны частично, что не дает гарантий обнаружения ботнета в обработанном объеме данных, даже при условии его передачи по каналу. Обработка же

всего объема передаваемых данных влечет за собой уменьшение пропускной способности канала до значения пропускной способности средства обнаружения распространения ботнетов, либо приводит к запозданию решения о вредоносности данных. Последнее означает, что даже если в проходящем трафике были обнаружены шеллкоды, то данные уже были переданы далее, а ущерб, возможно, уже нанесен.

Кроме того, для средства обнаружения и фильтрации шеллкодов крайне важен показатель ложных срабатываний. При больших объемах анализируемых данных, даже незначительный процент ложных срабатываний метода влечет за собой значительное абсолютное значение ложных алертов, что может сказаться на качестве обслуживания легитимных пользователей.

Апробация работы. Основные результаты диссертационной работы опубликованы в статьях [109], [110], [53], [111], [112], в том числе две [109] и [110] - в изданиях, рекомендованных ВАК для публикации результатов кандидатских и докторских диссертаций. Результаты докладывались на научном семинаре лаборатории вычислительных комплексов кафедры АСВК факультета ВМиК МГУ им М. В. Ломоносова под руководством член-корр. РАН Р. Л. Смелянского; на семинаре кафедры АСВК под руководством член-корр. РАН Л. Н. Королева; на научном семинаре группы «Network and system security» исследовательского подразделения компании Майкрософт, а так же на следующих конференциях:

1. Конференция «РусКрипто», Солнечногорск, Россия, 27-29 марта 2011г.
2. Конференция «РусКрипто», Солнечногорск, Россия, 28-31 марта 2012г.
3. Международная конференция «DEFCON-20», Лас-Вегас, США, 26-30 июля 2012г.
4. Международная конференция «BlackHat-EU-13», Амстердам, Нидерланды, 12-15 марта 2013г.

5. Летний коллоквиум молодых ученых в области программной инженерии «SYRCoSE», Казань, Россия, 30-31 мая 2013г.
6. Международная конференция «NOPCON», Стамбул, Турция, 6 июня 2013г.

Работа была выполнена при поддержке фонда «Сколково».

Работа структурирована следующим образом. В главе 1 приводится математическая модель, в рамках которой ставится задача распознавания объектов - отнесения анализируемых объектов к множеству заданных классов, а так же предлагается решение поставленной задачи. Главы 2 и 3 посвящены обоснованию применимости предложенной модели к шеллкодам. В главе 2 описаны выделенные признаки шеллкодов, а так же предлагается классификация шеллкодов. В главе 3 приведен обзор существующих алгоритмов обнаружения шеллкодов. В главе 4 приведено описание инструментального средства обнаружения шеллкодов Demorpheus и приведены результаты его экспериментального исследования. Глава 5 содержит заключение представленной работы.

Глава 1. Задача распознавания объектов

В данной главе рассматривается формальная модель процесса распознавания шеллкода (объекта), свойства этого процесса, его основные понятия и их свойства. В рамках построенной модели ставится задача распознавания объектов - задача отнесения объектов в некоторому фиксированному множеству классов объектов. Так же в данной главе приводится доказательство существования решения задачи распознавания шеллкодов в рамках приведенной модели и приводится алгоритм, строящий такое решение.

1.1. Математическая модель

Рассмотрим набор некоторый набор битовых исполнимых строк $\{\mathfrak{s}_i\}_{i=1}^n = \{\mathfrak{s}_1, \ldots, \mathfrak{s}_n\}$, в дальнейшем называемых *исследуемыми объектами*, или просто *объектами*. Исполнимость битовых строк в данном случае будем понимать в интуитивном смысле этого слова.

Пусть множеству рассматриваемых объектов сопоставлен набор *признаков* $\{\mathfrak{f}_j\}_{j=1}^k$, каждым из которых обладает хотя бы один из объектов $\mathfrak{s}_i$ рассматриваемого множества. Пусть каждому объекту $\mathfrak{s}_i$ сопоставлено некоторое непустое подмножество $\{\mathfrak{f}_j\} \neq \varnothing$ признаков, которыми объект $\mathfrak{s}_j$ обладает. Введем на множестве объектов $\{\mathfrak{s}_i\}_{i=1}^n$ множество вычислимых функций $\{\mathfrak{F}_j(\mathfrak{s}_j)\}_{j=1}^k$, ставящих в соответствие объекту $\mathfrak{s}_i$ значение признака $\mathfrak{f}_j$, где признаки могут принимать значения из множества $\{0, 1\}$. Значение признака $\mathfrak{f}_j$ равно 0 в том случае, если объект $\mathfrak{s}_i$ не обладает признаком $\mathfrak{f}_j$ и 1 в противном случае.

Произведем разбиение множества исследуемых объектов на подмножества $\{K_1, ..., K_l\}$. В дальнейшем такие подмножества будем называть *классами* объектов. Будем считать, что каждый класс определяется некоторой структурой признаков из $\{\mathfrak{f}_k\}$. Такие подмножества будем называть

определяющими наборами для соответствующих классов. Каждый признак может входить в определяющие наборы нескольких классов. Некоторые признаки, описывающие класс, являются зависимыми друг от друга, то есть наличие одного признака $\mathfrak{f}_l$ предполагает наличие другого признака $\mathfrak{f}_m$. При этом стоит заметить, что данное отношение между признаками не обязательно является бинарным: наличие признака $\mathfrak{f}_m$ не обязано учитывать наличие признака $\mathfrak{f}_l$. Кроме того, будем считать, что часть признаков обязана присутствовать в определяющей структуре класса. Такие признаки будем называть *базовыми*. Признаки, которые *могут* присутствовать в определяющей структуре класса. будем называть *вариативными*. Признаки, наличие которых в определяющей структуре класса невозможно, будем называть *исключающими*. Взаимосвязь между признаками, определяющими некоторый класс K_j, и самим классом K_j, задается формулой $P_j(S) = [\mathfrak{F}_{j_{1_1}}(S) \vee \cdots \vee \mathfrak{F}_{j_{1_m}}(S)] \wedge \cdots \wedge [\mathfrak{F}_{j_{k_1}}(S) \vee \cdots \vee \mathfrak{F}_{j_{k_n}}(S)]$, представляемой в виде конъюнктивной нормальной формы (КНФ).

Каждому объекту $\mathfrak{s}_j$ сопоставим *информационный вектор* $\tilde{\alpha}(\mathfrak{s}_j) = (\alpha_1, \ldots, \alpha_l)$, кодирующий информацию о принадлежности объекта $\mathfrak{s}_j$ к классам $\{K_1, \ldots, K_l\}$. Бит вектора α_i принимает значение 1, если объект $\mathfrak{s}_j$ является объектом класса K_i, и 0 в обратном случае.

Пусть для множества функций $\{\mathfrak{F}(\mathfrak{s})\}$ задано отношение частичного порядка $\prec$, определяющее взаимосвязь между зависимыми признаками. Так как множество $\{\mathfrak{F}(\mathfrak{s})\}$ представляет из себя множество вычислимых функций, то для каждой функции $\mathfrak{F}_j \in \{\mathfrak{F}(\mathfrak{s})\}$ существует алгоритм $\mathfrak{A}_k$, реализующий ее. Будем считать, что на множестве алгоритмов $\{\mathfrak{A}\}$, так же сохраняется отношение частичного порядка $\prec$. Отношение частичного порядка в этом случае определяет, в какой последовательности должны быть запущены алгоритмы, вычисляющие функции зависимых признаков объектов. Кроме того, будем считать, что каждый алгоритм $\mathfrak{A}_k \in \{\mathfrak{A}\}$, при определении наличия признака $\mathfrak{f}_k$ может ошибаться, долю ошибок алгоритма

будем обозначать как $\mathfrak{P}_k$. Это может быть связано с тем, что вычисляемый признак в объекте может быть запутан - например, обфусцирован или зашифрован. Сложность работы алгоритма $\mathfrak{A}_k$ будем обозначать как $\mathfrak{c}_{\mathfrak{A}_k}$.

Рассмотрим теперь еще одну сущность - *классификатор*. Классификатор μ_i содержит структуру некоторого подмножества $\{\mathfrak{A}_k\}'$ алгоритмов $\mathfrak{A}_k$, представляющую из себя граф, в котором узлы являются алгоритмами из подмножества $\{\mathfrak{A}_k\}'$, а дуги соответствуют путям передачи входного объекта между алгоритмами. Все алгоритмы, входящие в классификатор μ_i структурированы с учетом отношения частичного порядка: алгоритмы, имеющие зависимость, организуются в цепочку. Пусть, например, классификатор μ_1 содержит алгоритмы $\mathfrak{A}_1, \mathfrak{A}_2, \mathfrak{A}_3, \mathfrak{A}_4$, причем $\mathfrak{A}_1 \prec \mathfrak{A}_2, \mathfrak{A}_1 \prec \mathfrak{A}_3, \mathfrak{A}_2 \prec \mathfrak{A}_4, \mathfrak{A}_3 \prec \mathfrak{A}_4$. В этом случае классификатор будет содержать граф, представленный на рисунке 1.1.

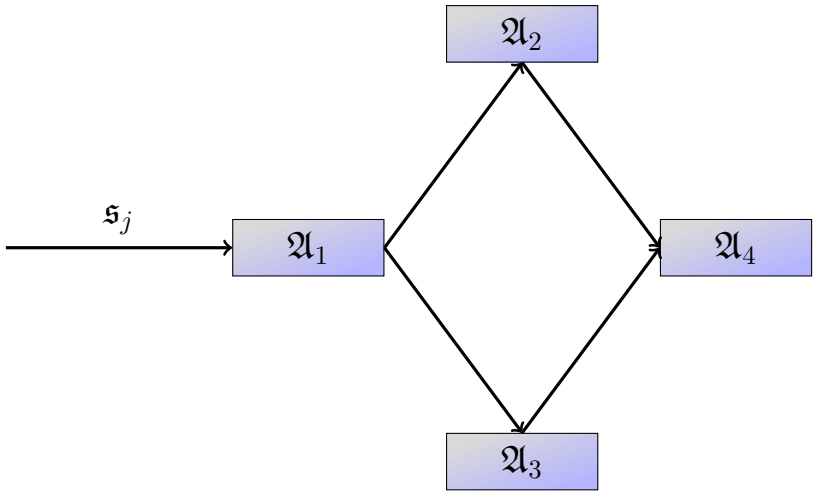


Рис. 1.1: Пример классификатора.

Будем считать, что классификатор μ_i способен распознавать объекты класса K_j в том случае, если пересечение множества признаков, вычисляемых алгоритмами, которые содержатся в классификаторе μ_i, и множества признаков, определяющих класс K_j, не пусто. Каждый классификатор характеризуется некоторой долей ложных срабатываний. Причины неоднозначности в распознавании классов классификатором две. Во-первых, как

уже отмечалось, сами алгоритмы из множества $\{\mathfrak{A}\}$ определяют признаки с некоторой долей ошибки, что оказывает непосредственное влияние на точность классификатора, в состав которого они входят. Во-вторых, классификатор может анализировать не все признаки, определяющие некоторый класс, а лишь их часть, что так же сказывается на точности классификатора.

В рамках построенной модели рассмотрим задачу распознавания объектов - задачу отнесения объектов к некоторым классам из заданного списка классов.

1.2. Задача распознавания объектов

Пусть задано фиксированное множество классов $\{K_l\}$ объектов. Пусть так же задано множество классификаторов $\{\mu_m\}$, способных распознавать объекты всех классов из множества $\{K_l\}$, при этом каждый классификатор распознает некоторое непустое подмножество классов $\{K_l\}$. Задача состоит в построении распознающего алгоритма $\mathfrak{A}'$, представляющего из себя такую суперпозицию классификаторов, для которой выполнено следующее условие:

- Для любого объекта $\mathfrak{s}_i$, принадлежащего любому классу K_j из множества заданных классов, структура классификаторов, построенная алгоритмом $\mathfrak{A}'$, распознает объект $\mathfrak{s}_i$ как объект класса K_j.

В качестве доказательства **существования решения** этой задачи укажем алгоритм построения.

Рассмотрим два классификатора $\mu_i = (\mathfrak{A}_1, \ldots, \mathfrak{A}_n, \prec_{\mathfrak{A}})$ и $\mu_i = (\mathfrak{A}_k, \ldots, \mathfrak{A}_{k+l}, \prec_{\mathfrak{A}})$. По определению классификаторов, алгоритмы, входящие в них, структурированы в соответствии с отношением частичного порядка, заданного на множестве алгоритмов. Так, $\mathfrak{A}_1 \leq \mathfrak{A}_i, i = 2 \ldots n$ и, аналогично, $\mathfrak{A}_t \leq \mathfrak{A}_i, i = k+1 \ldots k+l$. Будем считать, что $\mu_i < \mu_j$, если $\mathfrak{A}_1 < \mathfrak{A}_k$. Опера-

ции $=, >$ определяются аналогично. Таким образом, зададим на множестве классификаторов отношение частичного порядка $\prec_\mu$.

Структуру классификаторов будем строить как линейный список, отсортированный по возрастанию. Последнее необходимо для соблюдения зависимости между алгоритмами, входящими в один классификатор, и алгоритмами, входящими в другой, если такие имеются. Пример построенной структуры изображен на рисунке 1.2.

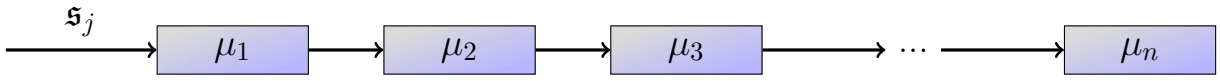


Рис. 1.2: Линейная структура классификаторов.

Исследуемый объект s_i поступает на вход очередному классификатору, внутри которого объект обрабатывается цепочкой алгоритмов, входящих в этот классификатор. В случае, если все алгоритмы в классификаторе выявили во входном объекте s_i соответствующие признаки, алгоритм $\mathfrak{A}'$ распознает объект s_i как принадлежащий к некоторому подмножеству классов $\{K'\}$, обнаруживаемых классификатором. После определения принадлежности объекта к некоторому подмножеству классов, меняются соответствующие биты информационного вектора объекта. Далее, объект передается на вход следующему классификатору.

Мы показали, что алгоритм $\mathfrak{A}'$, выстраивающий классификаторы в линейную структуру с сохранение частичного порядка алгоритмов, входящих в них, действительно является решением поставленной задачи в рамках заданной модели. □

Из существования решения поставленной задачи так же следует *корректность* построенной модели. Исследуем теперь это решение.

Покрытие классов объектов. В результате запуска структуры, построенной данным решением, для каждого объекта будет получен его информационный вектор, который, позволяет вычислить принадлежность

объекта к некоторому подмножеству классов. Распознавание объектов всего множества классов (полное покрытие классов объектов) возможно только в том случае, если для каждого класса K_i из заданного множества существует хотя бы один классификатор μ_j, алгоритмы которого вычисляют непустое подмножество признаков, определяющих класс K_i. Выполнение этого условия напрямую следует из формулировки задачи.

Доля ложных срабатываний решения. Как было отмечено ранее, каждый классификатор относит объект к классам с некоторой долей ложных срабатываний. В виду этого интересен для анализа вопрос о доле ложных срабатываний представленного решения. Будем считать, что распределение объектов различного вида (на которых классификатор может ошибаться или работать точно) - равномерное.

Рассмотрим цепочку из двух классификаторов μ_i и μ_j. Пусть событие A_1 соответствует ошибочному срабатыванию классификатора μ_i, а событие A_2 - ошибочному срабатыванию классификатора μ_j. Доли ложных срабатываний классификаторов обозначим как $P(A_1)$ и $P(A_2)$, соответственно. При этом будем считать, что события A_1 и A_2 независимы. Долю ложных срабатываний цепочки классификаторов обозначим за $P(A_1, A_2)$. Рассмотрим два возможных случая.

1 Пересечение множеств классов, определяемых классификаторами μ_i и μ_j, не пусто.

  В этом случае $P(A_1, A_2) = P(A_1)P(A_2)$. Учитывая, что $0 \leq P(A_1) \leq 1$ и $0 \leq P(A_2) \leq 1$, то $P(A_1, A_2) \leq min(P(A_1), P(A_2)$. Таким образом, общая доля числа ложных срабатываний монотонно убывает с увеличением числа классификаторов в цепочке.

2 Пересечение множеств классов, определяемых классификаторами μ_i и μ_j, пусто.

  В таком случае доли ложных срабатываний классификаторов не влияют друг на друга. Общая доля ложных срабатываний цепочки клас-

сификатора может быть оценена как $P(A_1, A_2) \leq max(P(A_1), P(A_2))$. То есть доля ложных срабатываний цепочки классификаторов будет не хуже, чем доля ложных срабатываний самого неточного классификатора.

Отсюда можно сделать следующий вывод. Классификаторы, определяющие пересекающиеся подмножества классов и структурированные в цепочку, существенно снижают долю ошибок структуры. В такой структуре каждый следующий классификатор "перепроверяет"результат работы предыдущего. Линейная структура в этом случае будет наиболее оптимальной с точки зрения доли ошибок.

Вычислительная сложность выполнения линейной структуры. Сложность построения линейной структуры классификаторов сводится к задаче сортировки элементов (в виду того, что классификаторы «сортируются» по возрастанию в соотвествии с заданным отношением частичного порядка). Для решения этой задачи существует множество подходов, имеющих приемлемую сложность - сортировка слиянием, быстрая сортировка и другие. Средняя сложность этих алгоритмов оценивается как $O(n\ log(n))$, где n - количество классификаторов.

Алгоритм построения структуры классификаторов запускается один раз. Далее построенная структура используется для распознавания объектов, поэтому сложность выполнения такой структуры имеет большое значение. Как было отмечено ранее, $(\mathfrak{c})_{\mathfrak{A}_\mathfrak{i}}$ обозначает сложность выполнения алгоритма $\mathfrak{A}_i$. Тогда сложность выполнения классификатора μ_i, в состав которого входят алгоритмы $\mathfrak{A}_{i_1}, \ldots, \mathfrak{A}_{i_k}$, будет определяться как $\mathfrak{c}_{\mu_i} = \sum_{j=i_1}^{i_k} \mathfrak{c}_{\mathfrak{A}_j}$. Так как в линейной структуре классификаторов при анализе входного объекта $\mathfrak{s}_j$ должен быть запущен каждый из классификаторов множества $\{\mu_i\}_{i=1}^m$, то сложность выполнения линейной структуры представляется соотношением $\mathfrak{C} = \sum_{i=1}^m \mathfrak{c}_{\mu_i} = \sum_{\forall j: \mathfrak{A}_i \in \{\mu_i\}_{i=1}^m} \mathfrak{c}_{\mathfrak{A}_i}$, то есть равно суммарной сложности классификаторов, входящих в структуру.

Итак, мы показали существование решения поставленной задачи распознавания в рамках построенной модели. Рассмотрим вопрос о существовании более оптимального по сложности выполнения решения задачи.

1.3. Постановка уточненной задачи распознавания объектов

Рассмотрим некоторый классификатор μ_i, в состав которого входят алгоритмы $\mathfrak{A}_{i_1}, \mathfrak{A}_{i_2}, \mathfrak{A}_{i_3}$. Пусть выполнено следующее условие: $\mathfrak{A}_{i_1} < \mathfrak{A}_{i_2} < \mathfrak{A}_{i_3}$, то есть алгоритм $\mathfrak{A}_{i_2}$ зависит от алгоритма $\mathfrak{A}_{i_1}$, а алгоритм $\mathfrak{A}_{i_3}$ зависит от алгоритма $\mathfrak{A}_{i_2}$. В этом случае, если алгоритм $\mathfrak{A}_{i_1}$ не выявил во входном объекте соответствующего признака, то запуск алгоритмов $\mathfrak{A}_{i_2}$ и $\mathfrak{A}_{i_3}$ не повлияет на результат работы классификатора. Аналогично, если алгоритм $\mathfrak{A}_{i_2}$ не выявил во входном объекте соответствующего признака, нет необходимости запускать алгоритм $\mathfrak{A}_{i_3}$.

В этом случае примем следующее допущение. Пусть каждый алгоритм $\mathfrak{A}_i$ может сбрасывать входящие в него объекты, то есть не передавать объект на вход другим алгоритмам, если он не обнаружил во входном объекте соответствующий признак. Кроме того, будем считать, что такой же функцией могут обладать и сами классификаторы: в случае, если классификатор μ_i не распознал входной объект как объект обнаруживаемых им классов, классификатор может не передавать объект на вход другим классификаторам. С учетом результатов, полученных при анализе решения задачи, описанной в разделе 1.2, рассмотрим более оптимальную задачу распознавания объектов.

Уточненная задача распознавания объектов. Пусть задано фиксированное множество классов $\{K_1, \ldots, K_l\}$ объектов. Пусть так же задано множество классификаторов $\{\mu_1, \ldots, \mu_m\}$, способных распознавать объекты всех классов их множества $\{K_1, \ldots, K_l\}$, при этом каждый классификатор распознает непустое подмножество заданных классов. Кроме того, будем

считать, что классификаторы могут сбрасывать объекты. Задача состоит в построении распознающего алгоритма $\mathfrak{A}''$, представляющего из себя суперпозицию классификаторов, такую, что выполнены следующие условия:

- Для любого объекта $\mathfrak{s}_i$, принадлежащего любому классу K_j из множества заданных классов, структура классификаторов, построенная алгоритмом $\mathfrak{A}'$, распознает объект $\mathfrak{s}_i$ как объект класса K_j. Другими словами, выполняется требование полноты покрытия заданных классов объектов.
- Полученная структура классификаторов не ухудшает значения доли ложных срабатываний: $\mathfrak{P} \leq max\mathfrak{P}_{\mu_i}$.
- Полученная структура оптимизирует суммарную вычислительную сложность входящих в нее алгоритмов: $\mathfrak{C} \leq \sum_i \mathfrak{c}_{\mathfrak{A}_i}$, где $\{\mathfrak{A}_i\}$ - множество алгоритмов, входящих в множество классификаторов $\{\mu_i\}$.

1.4. Предлагаемый подход к решению задачи распознавания

Организуем классификаторы в направленный ориентированный *граф* G, множество вершин $\{V\}$ соответствует классификаторам непосредственно, а множество ребер $\{E\}$ соответствует передаче данных между классификаторами. Граф G является частным случаем дерева принятия решений [108].

Граф G будем строить таким образом, чтобы передача потоков данных осуществлялась только между классификаторами, пересечение обнаруживаемых классов объектов которых не пусто. Именно в таком случае, как было показано в разделе 1.2, имеет место «перепроверка» результатов работы предыдущего классификатора, что приводит к снижению доли ложных срабатываний структуры классификаторов. Поступление входных объектов на классификаторы, пересечение обнаруживаемых классов которых пусто, должно осуществляться независимо.

Возможная структура такого графа приведена на Рисунке 1.3.

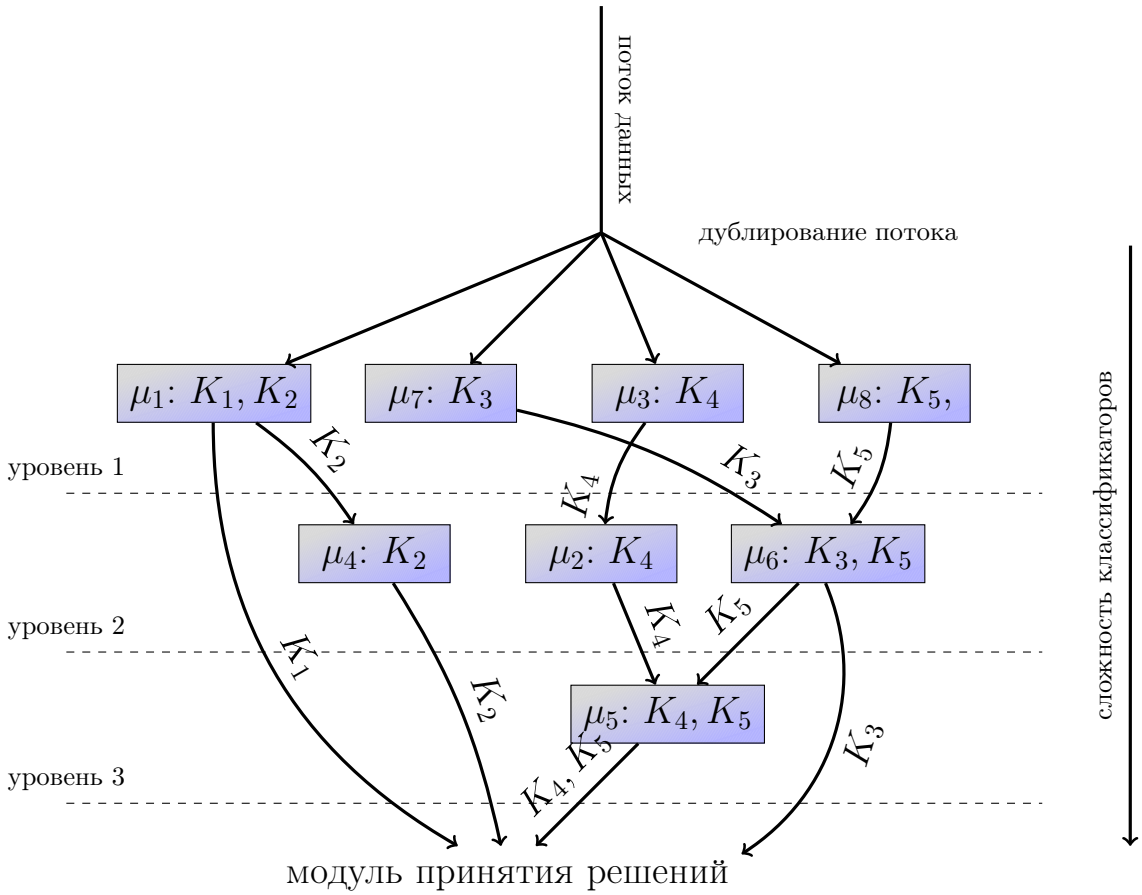


Рис. 1.3: Возможный пример топологии графа принятия решений.

При такой структуре классификаторов, анализируемый поток данных, состоящий из множества объектов $\{\mathfrak{s}_j\}$ поступает одновременно на классификаторы, обнаруживающие различные классы объектов. *Уровнем i графа* G назовем такой набор классификаторов, для которых выполнены следующие условия:

1. между классификаторами уровня i не осуществляется передача входных объектов;
2. классификаторы уровня i получают на вход объекты из классификаторов уровня $i-1$;
3. классификаторы уровня i передают входные объекты на классификаторы уровня $i+1$;

4. при условии того, что отработали классификаторы уровня $i-1$, классификаторы уровня i могут быть выполнены одновременно, то есть отсутствует зависимость по данным от всех низлежащих уровней.

Каждый уровень j графа G, содержащим всего J уровней, будем выбирать таким образом, чтобы он обеспечивал покрытие классов, обнаруживаемых классификаторами всех уровней $j \dots J$. Как следствие, на уровне $j = 1$ классификатора должно обеспечиваться полное покрытие множества $\mathfrak{M}$ классов шеллкодов. Кроме того, при выборе классификаторов уровня j необходимо, чтобы соблюдалось следующее условие: для любого классификатора μ_{j_p} уровня j, пересечение обнаруживаемых классов которого с любым классификаторов μ_{j-1_q}, должно выполняться $\mu_{j_p} >= \mu_{j-1_q}$. То есть между зависимыми классификаторами должно выполняться отношение частичного порядка. Поток данных, поступающий на первый уровень графа, дублируется между всеми классификаторами этого уровня.

Классификаторы, обладающие меньшей вычислительной сложностью, будем располагать на верхних уровнях графа G. Чем большей вычислительной сложностью характеризуется классификатор, тем на более низких уровнях он будет расположен. Как было отмечено ранее, классификаторы могут сбрасывать входные объекты, которые не были распознаны как объекты обнаруживаемых классов. Таким образом, объекты, не относящие к обнаруживаемым классам, будут отсекаться на верхних уровнях структуры, что приводит к снижению суммарной вычислительной сложности исполнения графа G. Объекты, ошибочно принятые некоторым классификатором за принадлежащий к обнаруживаемым им классом, будет передан на анализ другим классификаторам.

Итак, сведем поставленную задач распознавания объектов к задаче построения графа G классификаторов, обладающего следующими свойствами:

- Каждый уровень графа обеспечивает полное покрытие классов объ-

ектов, обнаруживаемых доступными для организации уровня классификаторов. Это значит, что если из множества классификаторов $\{\mu_i\}$ некоторое подмножество $\{\mu_j\}'$ составляет уровни $1 \dots j-1$, то для организации уровня j полнота покрытия классов должна обеспечиваться для классов, обнаруживаемых классификаторами из множества $\{\mu_i\} \backslash \{\mu_j\}'$;

- Ни один классификатор не встречается в графе более одного раза. Данное требование необходимо в связи с необходимостью оптимизации вычислительной сложности запуска графа G;
- Каждый уровень классификатора оптимален с точки зрения вычислительной сложности;
- Каждый уровень классификатора сохраняет отношение частичного порядка алгоритмов, входящих в классификаторы.

В качестве меры оптимальности для уровня графа в работе выбрано значение $\frac{\mathfrak{P}}{\mathfrak{C}}$, где $\mathfrak{P}$ - доля ложных срабатываний классификатора, а $\mathfrak{C}$ - сложность работы. При построении очередного уровня классификатора будем *максимизировать* значение этой меры. Выбор подобной меры обосновывается следующими рассуждениями. Как было отмечено выше, чем меньше вычислительная сложность классификатора, тем на более высоком уровне графа G он должен быть расположен. Этим обосновывается обратная зависимость сложности классификатора в выбранной мере. Если же принять гипотезу о том, что более сложные классификаторы обладают более высокой точностью работы, в то время, как более простые классификаторы характеризуются более высокой вероятностью ошибок, то вероятность ошибки классификатора должна быть прямо зависима в рассматриваемой мере.

Алгоритм построения графа классификаторов.

Рассмотрим эвристический алгоритм построения оптимальной топологии графа классификаторов.

На вход алгоритму поступает множество классификаторов, выходом алгоритма является топология графа классификаторов.

Описание алгоритма:

1. Если множество входных классификаторов алгоритма не пусто, перейти на шаг 2, иначе перейти на шаг 7.
2. Вычислить множество K классов шеллкодов, обнаруживаемых классификаторами из входного набора.
3. Построить все возможные комбинации из классификаторов входного набора, обеспечивающих покрытие классов из множества K и сохраняющие отношения частичного порядка с классификаторами предыдущего уровня.
4. Выбрать оптимальную комбинацию классификаторов. Данная комбинация является очередным построенным уровнем графа классификаторов.
5. Связать классификаторов из выбранной комбинации с последним уровнем графа. Связывание осуществляется в соответствии с обнаруживаемыми классификаторами классов.
6. Удалить классификаторы, входящие в оптимальную комбинацию из входного набора классификаторов. Вернуться на шаг 1.
7. Выход.

Листинг алгоритма представляется следующим образом:

Алгоритм 1: листинг на псевдо-языке

```
CONSTRUCT_GRAPH
input: set of classifiers M
return: graph G

 1:   previous_level = v;
 2:   G = previous_level;
 3:   K = RECOGNIZE_SHELLCODE_CLASSES(M);
```

```
4:   while M is not empty
5:          current_level = CREATE_NEXT_LEVEL(M,K);
6:          LINK_LEVELS(previous_level, current_level);
7:          for each m from current_level remove m from M;
8:          previous_level = current_level;

CREATE_NEXT_LEVEL
input: set of classifiers M, set of classes K
return: set of classifiers level for the constructed level
1:   arr = PROCESS_NEXT_LEVEL(M,K);
2:   level = OPTIMAL(arr);

PROCESS_LEVEL
input: set of classifiers M, set of classes K
return: array level of all possible classifiers combinations which
   provides coverage of K
1:   level = empty;
2:   for each m from M
3:          if CLASSES(m) != K
4:                 K_new = K/CLASSES(m);
5:                 M_new = REBUILD_SET(M/{m},K_new);
6:                 sub_level = PROCESS_NEXT_LEVEL(M_new,K_new);
7:                 if sub_level is not empty concatenate m with all
   elements from sub_level;
8:          else add m to the level;
9:          remove m from M;
10:         K = RECOGNIZE_SHELLCODE_CLASSES(M);

REBUILD_SET
input: set of classifiers M_new, detected classes K_new
return: set of classifiers M_new
1:   for each m from M_new
2:          if CLASSES(m) intersection K_new == 0
3:          remove m from M_new;
```

```
LINK_LEVELS
input: level_1, level_2
return: linked levels with respect to detected classes
1:   for each m_1 in level_1
2:          for each m_2 in level_2
3:                 if CLASSES(m_1) intersection CLASSES(m_2) is not empty
4:                        MAKE_LINK(m_1,m_2);
```

1.5. Исследование алгоритма

Рассмотрим вычислительную сложность описанного алгоритма. Предположим, что на вход алгоритму поступает множество M классификаторов, мощность которого равна $|M| = m$. Пусть входное множество M классификаторов способно обнаруживать множество K классов шеллкодов, мощность которого равна $|K| = k$. Для оценки вычислительной сложности процедуры CONSTRUCT_GRAPH, в первую очередь необходимо оценить вычислительную сложность всех вызываемых ею процедур.

Сложность шагов 1-2 алгоритма оценивается как

$$T(1-2) = O(1). \tag{1.1}$$

Вычислительная сложность вызываемой процедуры вычисления обнаруживаемых классов RECOGNIZE_SHELLCODE_CLASSES оценивается как

$$T(\text{RECOGNIZE\_SHELLCODE\_CLASSES}) \leq O(mk). \tag{1.2}$$

Такая оценка объясняется необходимостью анализа всех возможных классов шеллкодов для каждого классификатора из множества M. Сложность процедуры связывания построенного уровня с предыдущим уровнем графа классификатора LINK_LEVELS($level_1, level_2$) оценивается значением

$$T(\text{LINK\_LEVELS}(level_1, level_2)) \leq O(m_1 m_2) \leq O(m^2), \tag{1.3}$$

где $|level_1| = m_1, |level_2| = m_2$ и $m1 + m2 \leq m$ в общем случае. Сложность процедуры построения очередного уровня графа классификаторов CREATE_NEXT_LEVEL оценивается как

$$T(\text{CREATE\_NEXT\_LEVEL}) \leq T(\text{PROCESS\_LEVEL}) + O(m), \tag{1.4}$$

где $O(C_1 m) = O(m)$ - сложность выбора оптимальной комбинации при общем числе комбинаций C_1, являющимся некоторым константным значением. Анализируя вычислительную сложность процедуры PROCESS_LEVEL можно заметить, что

$$\begin{aligned} T(\text{PROCESS\_LEVEL}(4)) = T(\text{PROCESS\_LEVEL}(5)) \leq O(mk), \\ T(\text{PROCESS\_LEVEL}(7)) \leq 2O(m), \\ T(\text{PROCESS\_LEVEL}(8)) = T(\text{PROCESS\_LEVEL}(9)) \leq O(1). \end{aligned} \tag{1.5}$$

Таким образом, сложность процедуры PROCESS_LEVEL оценивается как

$$\begin{aligned} T(\text{PROCESS\_LEVEL}) = T(m,k) \leq m(2O(mk) + T(m',k') + \\ 2O(m) + 2O(1)) \leq mT(m',k') + O(m^2 k), \end{aligned} \tag{1.6}$$

где m' and k' - новые параметры рекурсивного вызова процедуры. Зафиксируем параметр k. Тогда, в худшем случае параметр m' будет иметь значение $m' = m - 1$. Следовательно,

$$T(m) \leq mT(m-1) + O(m^2) \approx O(m!m^2). \tag{1.7}$$

В среднем случае, параметр m' принимает значение $m' = \dfrac{m}{b}$, где b - эмпирически вычисленная константа. Таким образом,

$$T(m) \leq mT(m/b) + O(m^2). \tag{1.8}$$

Решение этого рекуррентного соотношения, согласно Основной теореме о рекурсии [34], принимает вид

1. $T(m) \leq O(m^2 logm)$, если $m = b^2$
2. $T(n) \leq O(m^2)$, если $m < b^2$
3. $T(n) \leq O(m^{log_b m})$, если $m > b^2$.

Предполагая, что b - эмпирически вычисленная константа с ограниченным значением, можно заметить, что начиная с некоторого $m > m_0$ всегда будет выполняться условие 3 Основной теоремы о рекурсии. Таким образом, освобождая параметр k, получаем следующую оценку для процедуры PROCESS_LEVEL в худшем случае:

$$T(\text{PROCESS\_LEVEL}) \leq O(km!m^2). \tag{1.9}$$

В среднем случае значение сложности выполнения процедуры принимает вид:

$$T(\text{PROCES\_LEVEL}) \leq O(km^{logm}). \tag{1.10}$$

Исходя из 1.1 - 1.10, можно сделать вывод, что сложность алгоритма построения оптимальной топологии графа в худшем случае представляется в следующем виде:

$$T(\text{CONSTRUCT\_GRAPH}) \leq m(O(mk) + O(m^2) + O(m!m^2k) + 3O(m) + 2O(1)) \approx O(m!k). \tag{1.11}$$

Сложность алгоритма построения оптимальной топологии графа в среднем случае представляется в виде:

$$T(\text{CONSTRUCT\_GRAPH}) \leq logm(O(mk) + O(m^2) + +O(km^{logm}) + 3O(m) + 2O(1)) \approx O(k \cdot logm \cdot m^{logm}). \tag{1.12}$$

Несмотря на сравнительно высокую вычислительную сложность алгоритма построения топологии классификатора, алгоритм остается применимым с практической точки зрения в виду нескольких допущений. В первую очередь, будем считать, что число классификаторов, поступающих на вход,

ограничено. Кроме того, алгоритм построения оптимальной топологии графа классификаторов запускается однократно перед началом анализа входных объектов, либо при добавлении новых классификаторов для перестроения топологии графа классификаторов. Исследуем теперь представленное решение.

Полнота покрытия классов. По построению графа G, его первый уровень должен включать в себя классификаторы $\{\mu_1, \ldots, \mu_m\}$ такие, что $\{K_1, \ldots, K_l\} \in \{K(\mu_1)\} \cup \cdots \cup \{K(\mu_m)\}$, где $\{K(\mu_i)\}$ - множество классов, обнаруживаемых классификатором μ_i. Другими словами, классификаторы первого уровня графа должны распознавать объекты всех классов $\{K_1, \ldots, K_l\}$. Таким образом, для $\forall K_i \exists \mu_j : K_i \in \{K(\mu_j)\}$.

Если на любом нижележащем уровне существует некоторый классификатор μ_p, так же обнаруживающий класс K_i, то между классификатором μ_j и μ_p будет существовать путь в графе G. Если же на нижележащих уровнях такие классификаторы отсутствуют, то ребро из классификатора μ_i будет входить в модуль принятия решений непосредственно. Отсюда можно сделать вывод, что граф G обеспечивает полное покрытие классов $\{K_1, \ldots, K_l\}$.

Доля ложных срабатываний. Ориентированный граф G можно представить в виде набора линейных структур классификаторов (цепочек), в которых на позиции i расположены классификаторы уровня i или модуль принятия решений. Будем считать, что при анализе доли ложных срабатываний выполнения графа G модуль принятия решений роди не играет. Любой классификатор, в который входит n ребер, будет включен в n таких линейных структур. Обозначим линейные структуры классификаторов за $L_1, \ldots, L_q$, где q - количество классификаторов первого уровня в графе G.

Для каждой линейной структуры L_i, проводя рассуждения, аналогичные анализу доли ложных срабатываний для решения, описанного в 1.2, получим, что $P(L_i) \leq min(P(\mu_{i_1}), \ldots, P(\mu_{i_k})$, где $\mu_{i_1}, \ldots, \mu_{i_k}$ - классифика-

торы, входящие в линейную структуру L_i.

Суммарная доля ложных срабатываний работы графа G оценивается соотношением $P(G) \leq max(P(L_1), \ldots, P(L_q) \leq max(min(P(\mu_{1_1}, \ldots, P(\mu_{1_w})), \ldots, min(P(\mu_{q_1}, \ldots, P(\mu_{q_r}))))$. Обозначим за μ^{L_i} классификатор, обладающий минимальной долей ложных срабатываний в цепочке L_i. Тогда $P(G) \leq max(P(\mu^{L_1}). \ldots, P(\mu^{L_q})) \leq max(P(\mu_i)))$.

Анализ вычислительной сложности выполнения. Снижение суммарной вычислительной сложности выполнения графа G достигается за счет размещения наиболее простых классификаторов на верхних уровнях графа и за счет возможности классификаторов сбрасывать объекты, не принадлежащие к обнаруживаемым классификатором классов.

Рассмотрим два возможных случая.

1. Входной объект $\mathfrak{s}_j$ принадлежит всем заданным классам $\{K_1, \ldots, K+l\}$. В этом случае должны быть запущены все классификаторы графа G, так как входной объект сбрасываться не будет. В таком случае $\mathfrak{C}(G) = \sum_{i=1}^{m} \mathfrak{c}_{\mu_i}$.
2. В среднем ожидаемом случае входной объект $\mathfrak{s}_j$ принадлежит некоторому подмножеству входного множества классов. В этом случае, при анализе объекта $\mathfrak{s}_j$ будет выполнена только часть графа G - некий подграф $G^{'}$. Тогда, $\mathfrak{C}(G^{'}) < \sum_{i=1}^{m} \mathfrak{c}_{\mu_i}$, так как не все классификаторы будут запущены.

Обобщая оба этих случае, получим, что $\mathfrak{C}(G) \leq \sum_{i=1}^{m} \mathfrak{c}_{\mu_i}$.

Итак, мы доказали, что построенный граф G удовлетворяет требованиям поставленной задачи. Покажем теперь, что представленная в текущей главе модель применима к исследуемой в работе задаче обнаружения шелл-кодов.

Глава 2. Классификация вредоносного исполнимого кода

Рассмотрим теперь исследуемую область в терминах модели, введенной в главе 1. Множеством объектов $\{\mathfrak{s}_j\}$ будем считать набор исполнимых строк, полученных путем *дизассемблирования* байтовых строк. К примеру, такие строки могут быть получены из TCP-сессии, из PDF-файлов, из java-script и других исследуемых на предмет вредоносности сущностей.

Как было отмечено в главе 1, множеству объектов сопоставлен набор признаков $\{\mathfrak{f}_j\}_{j=1}^{k}$. В данном разделе выделено множество признаков, характерных для шеллкодов. В зависимости от эксплуатируемых уязвимостей, шеллкоды могут обладать некоторыми специфичными признаками, характерными только для данной уязвимости. Кроме того, вид шеллкода может меняться так же в зависимости от вида защиты памяти на уровне операционной системы (stack canaries [44], рандомизация адресного пространства [104], запрет на выполнение команд в стеке), которую шеллкодам приходится обходить для успешного запуска вредоносной нагрузки. Шеллкод, обходящий некоторый вид защиты памяти на уровне операционной системы и эксплуатирующий некоторую уязвимость, обладает некоторым набором признаков, отличающих его от шеллкодов, обходящих другие виды защиты и/или эксплуатирующих другие уязвимости. Однако стоит так же заметить, что часть признаков может быть присуща и легитимным объектам. К таким признакам, например, можно отнести те, с которые отличают исполнимый код от случайного набора данных.

Итак, первым шагом на пути решения поставленной задачи является выявление списка признаков шеллкодов. При выделение признаков были использованы литературные источники - материалы, описывающие способы эксплуатации уязвимостей работы с памятью вредоносным исполнимым кодом и соответствующих ограничений, накладываемых на такой код. Так

же были рассмотрены материалы, описывающие механизмы обнаружения вредоносных исполнимых кодов. Кроме того, при выделении признаков использовался анализ исходных кодов шеллкодов, содержащихся в доступных базах эксплойтов. Примером базы публично опубликованных вредоносных кодов может служить база exploitdb [47]. Все признаки, полученные при анализе исходных кодов, были выделены вручную (на момент написания работы источников, содержащих описание признаков шеллкодов не существует).

На основе выделенных признаков в данной главе предлагается разбиение пространства шеллкодов на классы $\{K_1, \ldots, K_l\}$.

Данная глава организована следующим образом. В разделе 2.1 описан процесс выделения признаков шеллкодов и приведен список выделенных признаков, в разделе 2.2 предложена классификация шеллкодов.

2.1. Признаки вредоносного исполнимого кода

В процессе чтения научных статей, посвященных вредоносному исполнимому коду, а так же анализа исходных кодов (в случае его доступности) и изучения принципов их работы, был выявлен набор признаков шеллкодов. Такие признаки классифицируются по следующим критериям:

1. **Универсальность признака.** Критерий позволяет классифицировать признаки вредоносного исполнимого кода, эксплуатирующего ошибки работы с памятью, в зависимости от того, является ли признак общим для всех образцов вредоносного исполнимого кода, или же этот признак специфичен для конкретного образца или некоторого семейства образцов.
    - Универсальные признаки. К этой группе относятся признаки, характерные для всех образцов вредоносного исполнимого кода, эксплуатирующего уязвимости работы с памятью. Стоит отме-

тить, что такие признаки могут быть и у легитимных объектов. Это связано с тем, что, в первую очередь, исследуемый вредоносный исполнимый код представляет из себя программу, осуществляющую какую-либо активность. Ввиду этого, выделяется набор признаков характерных для исполнимых программ. Само по себе наличие подобных признаков во входном потоке данных не доказывает его вредоносность, однако позволяет отличить исполнимый код от набора случайных данных. Входной поток, обладающий подобными признаками может потенциально являться шеллкодом.

В то же время, исследуемый вредоносный исполнимый код имеет ряд ограничений, например, по размеру. Таким образом, возможно выделить ряд признаков, характерных для всех образцов шеллкодов.

- Специфичные признаки. К этой группе относятся признаки, характерные для конкретного образца шеллкода или для некоторых семейств, но не всех одновременно.

Оправданность поиска универсальных признаков во входном потоке данных связана, в первую очередь, с тем, что в реальных каналах бóльшая часть данных, передаваемая по трансмагистральным каналам, не является исполнимым кодом. Универсальные признаки позволяют отличить случайные данные от исполнимого кода, который может потенциально являться шеллкодом.

2. **Способ выявления признака.** По способу выявления признаки классифицируются, исходя из того, каким типом анализа (статическим, динамическим) возможно выявление этого признака в исследуемых данных. По данному критерию признаки вредоносного исполнимого кода классифицируются на следующие:

   - **Статические**. К этой группе относятся признаки, наличие кото-

рых в исследуемом объекте можно выявить путем статического анализа объекта - анализа кода объекта без его непосредственного исполнения.

- **Динамические**. К этой группе относятся признаки, наличие которых в исследуемом объекте возможно выявить путем непосредственного исполнения кода объекта.

3. **Платформа**. Шеллкод может изменяться в зависимости от платформы, на которой запущено приложение, эксплуатируемое шеллкодом. В виду того, что шеллкодами используются специфичные особенности операционных систем, особенности реализации команд процессора, то шеллкод, написанный под одну платформу, в общем случае не запускается корректно под другими платформами. Данный критерий определяет платформу, под которой выполняется вредоносный исполнимый код. Признаки такого кода для разных платформ могут не только не совпадать, но так же и противоречить друг другу: признаки, характерные для шеллкодов под одну платформу, могут являться обоснованием того, что рассматриваемый код не является шеллкодом под другую платформу.

   - **x86**. К этой группе относятся признаки, наличие которых выявляется в шеллкодах, исполняющихся на целевом процессоре x86.
   - **ARM**. К этой группе относятся признаки, наличие которых выявляется в шеллкодах, исполняющихся на целевом процессоре семейства ARM [80]. Процессор ARM может работать в специальном режимом Thumb [56], в котором используется сокращенная система команд. В стандартном режиме процессора ARM разрядность используемых команд равна 32 [81], в то время, как режим Thumb использует 16-битные команды. С помощью анализа исходных кодов шеллкодов, написанных под платформу ARM, было выявлено, что злоумышленники зачастую используют технику

переключения режима процессора в шеллкоде. Это позволяет записать большее количество команд в ограниченный по размеру шеллкод, а так же уклониться от обнаружения простыми сигнатурными детекторами. Начиная с версии ARMv6T2 процессора, доступна технология Thumb-2, позволяющая добавлять 32-битные инструкции.

В результате проведенного анализа было выделено четырнадцать статических и пять динамических признаков, присущих шеллкодам. Результирующие значения критериев признаков приведены в таблицах 2.1 и 2.2.

2.1.1. Статические признаки

1. *Дизассемблирование входных данных в цепочку инструкций определенной длины.* Очевидно, что подобное свойство можно обойти, например, с помощью техники самораспаковывания - получения новых инструкций и записи их в память в процессе непосредственного исполнения кода. Тем не менее, в ходе экспериментального исследования существующих шеллкодов, приведенного в работе [99], было показано, что для выявления программой вредоносных свойств, количество ее инструкций должно превосходить значение 14. Исполнимый код, включающий в себя меньшее количество инструкций, как правило, является последовательностью случайных данных, реже - легитимной программой. Признак является универсальным для всех платформ. Признак был выделен путем анализа литературных источников.
2. *Дизассемблирование данных в команды ARM и в команды режима Thumb.* Так как у процессора ARM существует два режима работы, некорректное дизассемблирование входного потока в команды одного из режимов может означать, что произошла смена режима работы процессора. В этом случае, дизассемблированный байтовый поток бу-

Признак	Универсальность	Платформа
Корректное дизассемблирование	Да	x86, ARM
Корректное дизассемблирование в команды ARM и Thumb	Да	ARM
Наличие команды смены режима процессора	Да	ARM
Корректное дизассемблирование с каждого смещения	Да	x86, ARM
Наличие GetPC	Нет	x86
Число push-call паттернов	Да	x86
Загрузка значений в регистры перед системным вызовом	Да	ARM
Диапазон адресов возврата	Нет	x86, ARM
Использование инвариантов	Нет	x86
Длина MEL	Нет	x86, ARM
Длина цепочек IFG	Да	x86, ARM
Размер объекта	Да	x86, ARM
Вид последней инструкции	Нет	x86, ARM
Инициализация операндов	Нет	x86, ARM

Таблица 2.1: Значения критериев статических признаков.

Признак	Универсальность	Платформа
Количество чтений полезной нагрузки	Нет	x86
Количество уникальных записей в память	Нет	x86
Передача управления на другой адрес	Нет	x86, ARM
Количество wx-инструкций	Нет	x86
Соответствие сигнатуре в зависимости от флагов	Нет	ARM

Таблица 2.2: Значения критериев динамических признаков.

дет корректно выполнен на целевом процессоре, несмотря на то, что с точки зрения статического анализа, он будет представлять из себя последовательность случайных данных. Подобная техника широко используется злоумышленниками в современных эксплойтах, позволяя избегать обнаружение шеллкода статическими детекторами, а так же исполнить большее количество инструкций. Последнее свойство актуально в виду ограниченности размера шеллкода. Признак является универсальным для платформы ARM. Признак был выделен путем анализа доступных исходных кодов шеллкодов.

3. *Наличие команды смены режима процессора.* Последней командой дизассемблируемого байтового потока в режиме ARM (или Thumb) перед началом случайных данных, должна быть команда `bx Rm`, переводящая процессор в другой режим работы. Здесь `Rm` - регистр общего назначения. Признак должен проверяться в случае наличия в анализируемом потоке предыдущего признака: дизассемблируемости различных частей байтового потока в инструкции двух режимов процессора. Признак является универсальным для платформы ARM. Признак был выделен путем анализа доступных исходных кодов шеллкодов.
4. *Дизассемблирование входных данных с каждого смещения.* Как было отмечено в Главе , шеллкод, эксплуатирующий уязвимость переполнения стека, перезаписывает адрес возврата уязвимой функции на некоторый адрес, указывающий в NOP-след. В общем случае злоумышленник не знает особенностей атакуемого узла, от которых может зависеть абсолютный адрес начала NOP-следа в адресном пространстве исполнимого процесса. Тем не менее, целью злоумышленника является выполнение вредоносной нагрузки объекта не зависимо от таких особенностей. Как следствие, NOP-след должен корректно дизассемблироваться с каждого смещения. Признак является специфичным для вредоносных исполнимых инструкций, содержащих NOP-след. При-

знак является универсальным для всех платформ. Признак был выделен путем анализа доступных исходных кодов шеллкодов.

5. *Наличие GetPC кода.* GetPC (Get Program Counter, так же известен как GetEIP) - набор исполнимых инструкций, вычисляющих свое расположение в адресном пространстве исполнимого процесса. GetPC код, как правило, необходим, чтобы возможно было подменить значение программного указателя на адрес самого кода непосредственно. Этот признак специфичен для вредоносных исполнимых инструкций, использующих техники самодекодирования и самомодификации. Признак не выполняется для ARM, как как в этом случае процессор поддерживает команду непосредственного получения значения программного счетчика. Признак был выделен путем анализа доступных исходных кодов шеллкодов.
6. *Число push-call паттернов превышает предопределенное пороговое значение.* Так как исследуемый объект является исполнимым кодом, он обладает характеристиками, специфичными для различных операционных систем. В частности, исполнимый код должен работать с вызовами операционной системы или библиотеки ядра. При выполнении кода, инструкции `call`, как правило, предшествует одна или несколько инструкций `push`. Пороговое значение `push-call` паттернов, равное 2, вычислено эмпирически в работе [102] . Признак не выполняется для платформы ARM, так как в этом случае аргументы вызова заносятся не в стек, а в регистры общего назначения. Кроме того, в ARM возможно занести значения сразу в несколько регистров вызовом одной инструкции. Данный признак позволяет отличить исполнимый код от набора случайных данных, часто встречающихся в канале. Признак является универсальным для всех платформ. Признак был выделен путем анализа литературных источников.
7. *Системному вызову `svc 0` предшествует загрузка некоторых зна-*

чений в регистры общего назначения, а количество системных вызовов превышает определенный порог. Признак вызван особенностью платформы ARM передавать локальные переменные функции через регистры, а не через стек. В частности, так же должно выполняться следующее условие: в регистр общего назначения `R7` должно быть занесено значение номера корректного системного вызова. Признак является универсальным для платформы ARM. Признак позволяет отличить исполнимый код платформы ARM от случайных данных. Признак был выделен путем анализа доступных исходных кодов шеллкодов.

8. *Адрес возврата находится в определенном диапазоне значений.* В шеллкоде адрес возврата перезаписывается значением, которое находится в диапазоне адресного пространства исполнимого процесса. Нижняя граница диапазона определяется адресом начала перезаписываемого буфера. Верхняя граница определяется как $RA - SL$, где RA – адрес поля адреса возврата и SL – длина шеллкода, или как $BS - SL$, где BS - адрес начала стека. Этот признак общий для всех исследуемых объектов, не использующих технику рандомизации стека ASLR [104]. Признак сохраняется для всех рассматриваемых платформ и выделен путем анализа доступных исходных кодов.
9. *Использование шаблонов.* В шеллкодах часто используются зарезервированные ключевые слова или априорно известные числовые константы. Такие признаки являются специфичными для вредоносных объектов, использующих конкретные уязвимости. В большинстве случаев, уязвимая программа содержит несколько путей выполнения. Какой путь будет выполняться зависит от поступивших на вход программе или некоторой ее функции данных. Таким образом, для того, чтобы выполнить нужный путь уязвимой программы, шеллкод содержит так называемые шаблоны структуры - некоторые зарезервированные клю-

чевые слова, расположенные в определенных позициях шеллкода. В таким ключевым словам, например, могут относиться части сетевого протокола в том случае, если уязвимое приложение анализирует пакеты. Шеллкоды, содержащие подобные шаблоны в своей структуре, Такие инварианты, были использованы при распространении Apache-Knacker [29] и сетевого червя Lion [60]. Признак выделен путем анализа литературных источников и доступных исходных кодов.

10. *Максимальная длина исполнимой цепочки (MEL - Maximum Execution Length) превышает определенное пороговое значение.* Длина исполнимой цепочки является важным признаком корректной последовательности инструкций, отличающего последовательность инструкций от случайных данных. Если цепочка заканчивается инструкцией перехода (например, `jmp`), то к значению прибавляется так же и длина цепочки, разобранной с адреса перехода. Пороговое значение, вычисленное экспериментально, составляет 256 байт [99]. Признак является общим для шеллкодов любых платформ. Тем не менее, данным признаком могут так же обладать и легитимные программы. Признак выделен путем анализа литературных источников.
11. *Граф IFG (граф потока инструкций), из которого удалены незначимые инструкции, содержит цепочку, превосходящую определенное пороговое значение.* Незначимыми инструкциями называют инструкции, приводящие к аномалиям потока данных и не влияющих на ход выполнения программы. Аномалии потока данных подразделяются на три типа [58]:
    - DD-аномалии(Define-Define). Аномалии такого типа возникают в том случае, если какая-либо переменная была определена, а затем переопределена. Между этими действиями использования этой переменной не происходит.

- DU-аномалии(Define-Undefine). Аномалии такого типа возникают, когда какая-либо переменная была определена, после чего переменой было присвоено неопределенное значение без ее использования.
- UR-аномалии(Undefine-Reference). Аномалии такого типа возникают в том случае, когда какой-либо переменной было присвоено неопределенное значение, после чего она была использована.

Критерий позволяет дифференцировать исполнимые инструкции и случайные данные, которые содержат большое количество аномалий потока данных даже в случае их дизассемблирования [102]. Критерий является универсальным и выделен путем анализа литературных источников.

12. *Размер объекта не превосходит определенного значения.* Шеллкод, использующий уязвимости работы с памятью, изменяет поток управления атакуемой программы путем перезаписи либо адреса возврата из стека, либо указателя служебной структуры на адрес внедренного вредоносного кода или на адрес определенного места внутри системной библиотеки. Так как и стек, и служебные структуры имеют ограниченный размер, то размер шеллкода так же лимитирован. Критерий является универсальным для всех вредоносных объектов, эксплуатирующих уязвимости работы с памятью. Признак выполняется для любых платформ. Признак выделен путем анализа литературных источников и исходных кодов.
13. *Если последняя инструкция в цепочке заканчивается командой перехода с прямой или абсолютной адресацией, то адресом перехода может являться либо адрес библиотечного вызова, либо корректный номер прерывания.* Как правило, при эксплуатировании уязвимости задачей злоумышленника является не аварийное завершение запущенного процесса, а например, получения контроля над консолью

с уровнем доступа ядра или какой-либо другой цели. Таким образом, код вредоносного объекта должен передавать управление системным вызовам, которые могут быть доступны явным вызовом библиотеки или путем прямого прерывания. В первом случае, команды перехода (в частности, `jmp, call, ret, load, int`) должны сопровождаться базовым загрузочным адресом для библиотек: `0x40` для Linux и 0x77 для Windows. В другом случае, соответствующие номера прерываний – `int 0x80` для Linux и `int 0x2e` для Windows. Для ARM и платформы андроид в частности, соответствующие номера прерываний - ??. Признак выявлен путем анализа исходных кодов шеллкодов, доступных в публичных библиотеках эксплойтов.

14. *Операнды самомодифицирующегося кода и кода с косвенными переходами должны быть инициированы.* Критерий является универсальным для всех платформ.

2.1.2. Динамические признаки

Среди динамических признаков выделены следующие:

1. *Количество чтений полезной нагрузки превышает определенный порог.* Процесс расшифрования вредоносным объектом своей полезной нагрузки во время непосредственного исполнения приводит к множественному доступу к памяти. Эта область памяти является небольшой частью виртуального адресного пространства, которой сопоставлен уязвимый буфер. Такая эвристика предполагает, что количество чтений из входного буфера в произвольном коде достаточно мало, в то время как декриптор обращается к множеству различных областей памяти внутри него. Этот признак является общим для полиморфных вредоносных исполнимых инструкций для процессора x86. Для платформы ARM признак выполняться не будет, так как процессор

поддерживает команды одновременной загрузки данных в 16 регистров, что позволяет применить одну и ту же операцию модификации к 16-ти блокам данных одновременно, что используется в современных эксплойтах. Признак выделен путем анализа литературных источников и исходных кодов шеллкодов.

2. *Количество уникальных записей в память превышает определенный порог.* Признак является специфичным для полиморфных вредоносных исполнимых инструкций. Признак так же не выполняется для платформы ARM. Признак выделен путем анализа литературных источников и исходных кодов шеллкодов.
3. *Поток управления хотя бы один раз передается из адресного пространства входного буфера на адрес, по которому ранее осуществлялась запись.* Такой признак специфичен только для вредоносных исполнимых инструкций, обходящих техники обнаружения полиморфизма путем записи полезной нагрузки в другую область памяти (non-self-contained). Признак универсален для любых платформ. Признак выделен путем анализа литературных источников.
4. *Количество исполненных wx-инструкций превышает определенный порог.* Под wx-инструкциями понимаются инструкции, которые были записаны в адресное пространство процесса в ходе его выполнения. Такие инструкции являются неотъемлемой частью вредоносных исполнимых инструкций, обходящих признаки обнаружения полиморфизма. Признак выделен путем анализа литературных источников и исходных кодов шеллкодов.
5. *В зависимости от определенных значений флагов выполняется набор инструкций, удовлетворяющих вредоносной сигнатуре.* Признак характерен для платформы ARM. Связан с тем, что процессор в этом случае допускает условное выполнение инструкций: в зависимости от

разных значений флагов ряд инструкций может быть выполнен либо не выполнен, что существенно изменяет вид реально выполняемой программы. Признак выделен путем анализа исходных кодов шеллкодов.

2.2. Классификация вредоносного исполнимого кода

Любой вредоносный исполнимый объект, эксплуатирующий ошибки работы с память, представляет из себя комбинацию признаков - вредоносных и легитимных. К примеру, длинная последовательность nop-инструкций `0x90` часто встречается в таких объектах, однако такая цепочка может быть обнаружена и в легитимном коде и случайных данных. Используя уникальные признаки вредоносных объектов и легитимных объектов, возможно разбить пространство вредоносных объектов на семейства, или *классы* - набор экземпляров вредоносных объектов, базирующихся на схожих характеристиках или схожих паттернов исполнения. Вредоносные объекты разделяются на классы в зависимость от части объекта, которую признаки, идентифицирующие этот класс, представляют - классы, базирующиеся на признаках активатора, декриптора, полезной нагрузки или зоны адресов возврата. Ниже приводится полный список таких классов.

Классы, базирующиеся на признаках активатора:

1. K_{NOP-PL}. Класс вредоносных объектов, активатором которых является простейший NOP-след. Простейший NOP-след состоит из набора NOP (no-operation) инструкций `0x90`. NOP-инструкции не влияют на поток управления программы, только увеличивая программный счетчик. Пример объекта такого класса был продемонстрирован в работе [75].
2. K_{NOP-B}. Класс вредоносных объектов, активатором которых является однобайтный NOP-эквивалентный след. NOP-след может быть об-

фусцирован путем замены NOP-инструкций на однобайтные инструкции, которые не имею значимого эффекта и, для целей атакующего, эквивалентны NOP-инструкциям. К примеру, такой инструкцией может быть та, которая уменьшаем и увеличивает значение регистра, не используемого в полезной нагрузке вредоносного объекта; инструкции, которые устанавливают, а зачем очищают некоторый флаг; комбинация инструкций `push` и `pop`. Существующие средства автоматической генерации полиморфных вредоносных объектов используют такие инструкции для уклонения от обнаружения. Например, ADMmutate [61] использует такую технику со списком из 55 однобайтных NOP-эквивалентных инструкций, Metasploit Framework [10] расширяет список ADMmutate до 58 однобайтных инструкций.

3. K_{NOP-MB}. Класс вредоносных объектов, активатором которых является многобайтный NOP-эквивалентный след. Однобайтные NOP-эквивалентные инструкции так же могут быть заменены на многобайтные NOP-эквивалентные инструкции, которые так же не влияют на выполнение полезной нагрузки вредоносного объекта. Тем не менее, любые многобайтные NOP-эквивалентные инструкции не могут быть включены в такой набор, так как след должен корректно исполняться с любого смещения. Для того, чтобы избежать этого ограничения, в настоящее время применяется следующая техника: операнды многобайтных инструкций ограничиваются таким образом, чтобы их значения соответствовали кодам операций однобайтных инструкций или кодам операций других многобайтных NOP-эквивалентных инструкций. Пример такого NOP-следа приведен на рисунке 2.1. В данном примере, если передача управления передается на самый левый байт, будет выполнена следующая цепочка инструкций: `cmp $0x35, %al, sub $0x40, %al, ...`. Если же передача управления осуществляется на след со смещением 1, то будет выполнена другая цепочка: `xor, or,`

. . . .

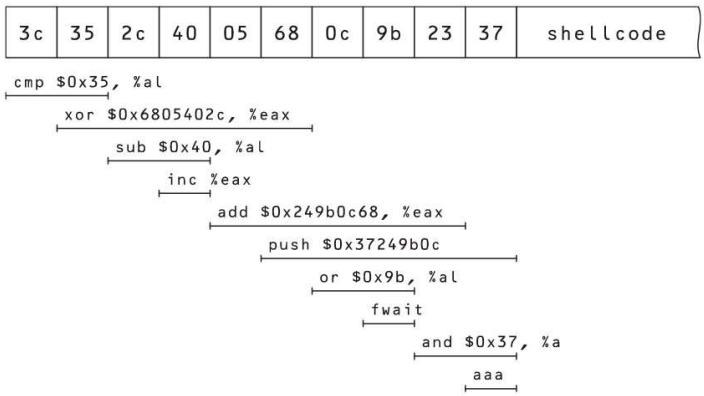


Рис. 2.1: Многобайтный NOP-эквивалентный след.

4. $K_{NOP-4AL}$. Класс вредоносных объектов, активатором которых является четырехбайтный выровненный след. Несмотря на то, что в общем виде NOP-след должен корректно дизассемблироваться и исполняться с каждого байтового смещения, выравнивание стека может ослабить это ограничение. По умолчанию, современные компиляторы выравнивают стек до размера слова (4 байта). Таким образом, возможно построить след, который должен быть исполнимым каждые 4 байта. Последовательности инструкций, не начинающихся со смещения, выровненного по слову, могут содержать любой тип инструкций, в том числе и некорректные.

5. K_{NOP-T}. Класс вредоносных объектов, активатором которых является батутный NOP-след. Типичный NOP-след передает контроль управления на полезную нагрузку вредоносного объекта путем последовательного прохождения инструкций. Та же функциональность может быть достигнута путем непосредственной передачи контроля управления по нужному смещению (рис. 2.2). Такие NOP-следы называются *батутными*.

   Тело такого шеллкода состоит из инструкций передачи управления

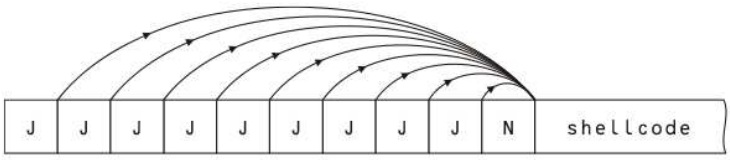


Рис. 2.2: Пример батутного NOP-следа.

с относительным адресом, указывающим непосредственно на начало полезной нагрузки. Таким образом, передача управления осуществляется за единственный шаг, с любого смещения. Батутный след так же может быть сгенерирован, основываясь на выравнивании по слову. В этом случае инструкции передачи управления помещаются по выровненным смещениям. Даже если NOP-след должен быть корректно исполнимым с любого смещения, возможно подобрать операнды инструкций передачи управления таким образом, чтобы их коды операций соответствовали NOP-эквивалентным инструкциям. Пример батутного следа, корректно исполнимого с каждого смещения, представлен на рисунке 2.3.

6. K_{NOP-TO}. Класс вредоносных объектов, активатором которых является обфусцированный батутный NOP-след. Так как число инструкций, которые могут быть использованы для генерации батутного NOP-следа, ограничено, такой NOP-след может быть обнаружен простыми сигнатурными методами. Для того, чтобы избежать обнаружения, батутный NOP-след может быть обфусцирован. Например, в след могут быть добавлены другие NOP-эквивалентные инструкции. В этом случае полезная нагрузка может достигаться в несколько шагов.
7. $K_{NOP-SAR}$. Класс вредоносных объектов, активатором которых является NOP-след, устойчивый к статическому анализу (SAR - static

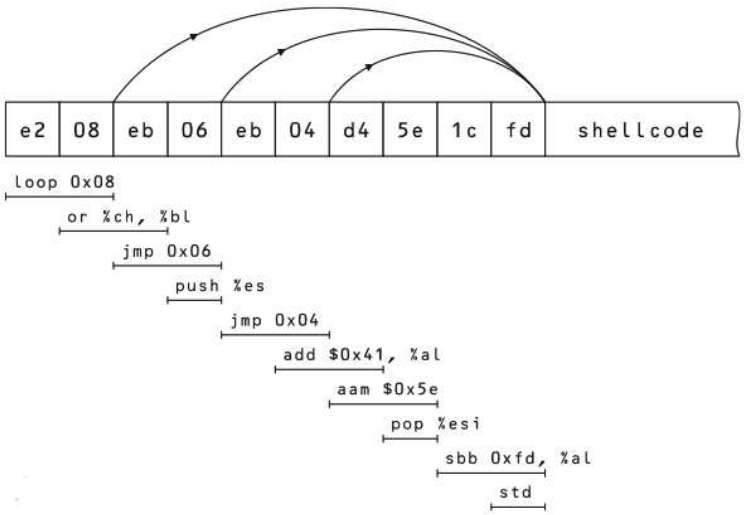


Рис. 2.3: Пример батутного NOP-следа, исполнимого с каждого смещения.

analysis resistant). Во время непосредственного исполнения такого NOP-следа, поведение инструкций контролируется злоумышленником. Таким образом, предсказать поведение инструкций, входящих в SAR NOP-след либо трудно, либо невозможно. Это достигается либо за счет использования условных инструкций, целевой адрес которых не может быть статично определен, либо за счет использования инструкций, организующих самомодифицирующийся код. Примером первого механизма может служить использование регистров или косвенных переходов. Кроме того, SAR-NOP-след может содержать так же и некорректные инструкции целевого процессора, так как они могут быть перезаписаны в процессе исполнения на корректные инструкции, после чего NOP-след будет успешно выполнен. Тем не менее, в этом случае след должен быть выровнен по размеру стека: необходимо избежать выполнения некорректных инструкций до их перезаписывания.

8. K_{GET-PC}. Класс вредоносных объектов, активатором которых является GetPC код - код, который определяет свое расположение в адрес-

ном пространстве исполнимого процесса. В этом случае управление на вредоносную полезную нагрузку исполнимых инструкций может быть передано непосредственно, без необходимости в NOP-следе.

Классы, базирующиеся на признаках декриптора.

1. K_{SU}. Класс самораспаковывающихся вредоносных объектов. Такие объекты изменяют, или распаковывают, блоки вредоносной полезной нагрузки, обфусцированной во время компиляции. Трансформации, примененные к полезной нагрузки могут быть как тривиально простыми (например, применение операции `xor` к блоку инструкций), так и достаточно сложными DEA (Data Encryption Algorithm) - алгоритмами (например, DES [86] и другие алгоритмы [87]).
2. K_{SC}. Класс зашифрованных вредоносных объектов.
3. K_{NSC}. Класс полиморфных вредоносных объектов, для которых выполнены два следующих условия:
   - Объект не содержит какой-либо формы GetPC кода;
   - Объект не выполняет операций чтения из своего собственного адресного пространства во время исполнения.

Классы, базирующие на обфускации полезной нагрузки вредоносного объекта.

1. K_{PL} Класс оригинальных вредоносных (необфусцированных) объектов.
2. K_{DATA} Класс вредоносных объектов, к которым была применена обфускация данных. Примером такой обфускации может служить замена ASCII символов на UNICODE символы.
3. K_R Класс вредоносных объектов, обфусцированных путем изменения порядка входящих в них инструкций. Таким способом возможно сгенерировать $n!$ различных объектов из исходного объекта, содержащего n инструкций.

4. $K_{ALT_{INS}}$ Класс вредоносных объектов, обфусцированных путем замены инструкций, входящих в объекты, другими инструкциями с той же самой операционной семантикой. К примеру, инструкция `xor` может быть заменена на инструкцию `sub`.
5. K_{INJ}. Класс вредоносных объектов, обфусцированных путем вставки "мертвого"кода - кода, на который никогда не будет передано управление.
6. K_{MET}. Класс метаморфных вредоносных объектов. При таких обфускациях используется два уровня метаморфизма: алгоритмический метаморфизм и метаморфизм уровня операционных кодов. Алгоритмический метаморфизм базируется на предположении, что некоторые блоки в алгоритме могут варьироваться. Метаморфизм уровня операционных кодов базируется на предположении, что каждый элемент псевдо-языка может быть заменен различными наборами кодов операций.

Классы, базирующиеся на зоне адресов возврата.

1. K_{RET}. Классы вредоносных объектов с инвариантными значениями зоны адресов возврата.
2. K_{RET_+}. Классы вредоносных объектов с обфусцированной зоной адресов возврата. К примеру, в таких объектах может быть изменен порядок следования младших и старших бит адреса. В этом случае, управление будет передано в другую область стека, однако условием является попадание в NOP-след.

Глава 3. Методы обнаружения вредоносного исполнимого кода

В данной главе приводится обзор существующих методов обнаружения шеллкодов. В терминах введенной в главе 1 модели, существующие методы обнаружения шеллкодов представляют из себя классификаторы $\{\mu_i\}_{i=1}^{m}$, в состав которых входит один или несколько алгоритмов, обнаруживающих признаки шеллкодов во входных объектах. Как будет показано, каждый из существующих методов обнаружения шеллкодов действительно обнаруживает некоторое подмножество классов шеллкодов. Так как методы, описанный в данной главе, могут включать в себя не все алгоритмы, определяющие тот или иной класс, что может привести к неоднозначности в распознавании объектов. В частности, в таком случае легитимный объект может быть ошибочно отнесен к некоторому классу шеллкодов, или шеллкод одного класса может быть ошибочно распознан как шеллкод другого класса. Такие ошибки будем называть *ошибками второго рода*, или FP (False Positives).

Кроме того, злоумышленниками активно применяются различные механизмы запутывания и обфускации шеллкодов, а так же самомодификации и шифрования для того, чтобы усложнить задачу обнаружения вредоносного исполнимого кода. В виду этого, метод обнаружения шеллкодов может не распознать шеллкод, как объект соответствующего класса. Такие ошибки будем называть *ошибками первого рода*, или FN (False Negatives).

Данная глава структурирована следующим образом. В разделе 3.1 описаны показатели эффективности методов обнаружения шеллкодов. В разделе 3.2 приведена классификация таких методов. В разделе 3.3 приведен обзор существующих методов обнаружения шеллкодов, в результате которого показана неприменимость рассматриваемых методов к реальным высокоскоростным каналам передачи данных и неспособность существую-

щих методов обеспечить полное покрытие классов шеллкодов.

3.1. Показатели эффективности методов

В данном разделе приведены показатели эффективности методов обнаружения вредоносного исполнимого кода.

1. **Доля ошибок первого рода**. Критерий оценивает точность обнаружения методами вредоносного исполнимого кода. Тем не менее, в исследуемой области не существует универсального набора данных для тестирования доли ошибок первого рода алгоритмов. Авторы предложенных методов, как правило, проводят свои исследования на специально сформированных наборах данных, на которых предложенные средства демонстрируют наилучшие результаты работы.
2. **Доля ложных срабатываний - ошибок второго рода**. Данный критерий оценивает долю ложных срабатываний методов. Высокоскоростные каналы передачи данных, на которых предполагается использование методов обнаружения шеллкодов, характеризуются большим объемом передаваемой информации в единицу времени. При большом объеме анализируемых данных даже незначительная доля подобных ошибок влечет за собой значительное абсолютное число ложных срабатываний. Большое число ложных срабатываний негативно сказывается на работе легитимных пользователей, уменьшая показатели качества их обслуживания, а в худшем случае - приводя к отказу в их обслуживании. Стоит отметить, что критерий оценивает долю ложных срабатываний на только тех образцах шеллкода, которые относятся к обнаруживаемым методом классам.
3. **Алгоритмическая сложность метода**. Данный критерий оценивает сложность методов, непосредственно влияющую на скорость их работы. Из-за необходимости анализировать значительное количество

данных, критерий является одним из наиболее критичных.

4. **Покрытие классов шеллкодов**. Критерий оценивает способность методов к обнаружению различных классов шеллкодов. Рассматриваемые классы шеллкодов представлены в Главе 2. В случае, если некоторый метод способен обнаружить все из приведенных классов, то будем считать, что метод обеспечивает полное покрытие классов шеллкодов. Если метод обнаруживает не все из перечисленных классов, то покрытие классов шеллкодов методов, соответственно, не полно.

3.2. Классификация методов обнаружения шеллкодов

Для классификации методов обнаружения шеллкодов предложены следующие критерии.

1. **Способ выполняемого анализа.** Критерий позволяет классифицировать методы обнаружения вредоносных исполнимых объектов в зависимости от необходимости эмуляции виртуального окружения.
2. **Обнаруживаемые участки шеллкода.** Критерий позволяет классифицировать методы обнаружения вредоносных исполнимых объектов в зависимости от того, какие участки типичной структуры шеллкода (активатор, декриптор, полезная нагрузка, зона адресов возврата) методы способны обнаруживать. Структура шеллкода подробно рассмотрена во введении.

По способу осуществляемого анализа, методы обнаружения вредоносных объектов подразделяются на следующие классы.

- *Методы статического анализа.* К этой группе относятся методы, осуществляющие анализ кода без его непосредственного выполнения.

  С теоретической точки зрения, с помощью статического анализа возможно рассмотреть все пути исполнения программы (для случая последовательных программ). Статический анализ, как правило, быст-

рее динамического. Методы статического анализа имеют ряд недостатков:

– Часть задач, относящихся к анализу поведения программы и ее свойств, в общем случае не разрешимы при использовании одного лишь статического анализа. В частности, в работе Эрика Филиола [49] доказан ряд следующих утверждений:

  Теорема 1. *Задача обнаружения метаморфного вредоносного объекта статическим анализом неразрешима.*

  Метаморфным считается шеллкод, полученный из оригинального образца одной из следующих обфускаций: алгоритмической обфускацией, предполагающей, что некоторые блоки в алгоритме могут варьироваться, либо обфускацией уровня кодов операций, предполагающая, что каждый элемент псевдо-языка (например, цикл) может быть заменен различными наборами инструкций (на примере цикла и языка C, можно использовать конструкцию `while`, `for`, а так же просто последовательное повторение тела цикла). Другими словами, метаморфной считается обфускация, сохраняющая только семантику программы, или семантическое ядро программы [49].

  Теорема 2. *Задача обнаружения полиморфного вредоносного объекта статическим анализом NP-полна. [49]*

  Полиморфным считается образец шеллкода, полученный из оригинального образца, путем применения обфускаций, изменяющих синтаксис программы [93].

– Возможно создание вредоносного кода, использующего техники уклонения от статического анализа. В частности, могут использоваться различные методы обфусцирования кода, косвенная адресация, самомодификация кода.

Ниже приведены широко применимые техники анализа кода, используемые в статических методах обнаружения вредоносных исполнимых кодов.

– *Сопоставление сигнатур.* Такой подход основан на сопоставлении анализируемого образца с сигнатурой – некоторым регулярным выражением. Сигнатурные методы подразделяются на методы, сопоставляющие с анализируемым кодом контекстно-зависимые сигнатуры - регулярным выражением, использующим исходных код шеллкода непосредственно, и методы, сопоставляющие сигнатуры поведения программ. В сигнатурах поведения учитывается возможность замены одних инструкций на другие со схожей семантикой: например, возможна взаимозаменяемость арифметических инструкций. Сигнатурный подход является неотъемлемой частью как достаточно простых, так и более усложненных алгоритмов обнаружения шеллкода. Некоторые методы являются частично-сигнатурными: они генерируют сигнатуры после того, как сделали заключение о вредоносности потока. Такие сигнатуры используются в дальнейшем с целью минимизации временных затрат на обнаружение уже исследованных образцов вредоносного кода. Существенный недостаток сигнатурного подхода - невозможность обнаружения полиморфной версии одного и того же образца. Методы, использующие технику сопоставления сигнатур, будем так же называть *сигнатурными.*

– *Анализ графов CFG/IFG.* CFG [52] (граф потока управления) - направленный граф, вершинами которого являются линейные блоки инструкций. Линейный блок - последовательность инструкций с одним входом, одним выходом и не содержащая инструкций перехода. Дуги GFG представляют из себя пути потока управления. В отличие от CFG, в IFG (граф потока инструк-

ций) [102] вершинами являются непосредственно инструкции.

  - *Анализ корректности дизассемблирования потока.* Методы, входящие в эту группу, осуществляют анализ на основе результатов процесса дизассемблирования. В частности, такие методы проверяют корректное дизассемблирование, начиная с каждого смещения входного буфера, а так же встречаемость привилегированных инструкций.
  - *Анализ на основе абстрактной интерпретации.* Абстрактная интерпретация - подход, при котором анализ изменения кода и доступности отдельных его блоков осуществляется с помощью предположения диапазона значений входных данных и переменных, влияющих на ход выполнения программы.

- *Динамические методы.* К этой группе относятся методы, анализирующие код программы в процессе ее непосредственного выполнения.

  В отличие от статических, динамические методы устойчивы к обфускации кода и различным методам уклонения от статического анализа (в том числе самомодификации). Тем не менее, динамические методы так же имеют ряд недостатков:

  - требуют гораздо больше накладных расходов, чем статические. В частности, может потребоваться выполнение достаточно длинной цепочки инструкций, прежде чем можно будет сделать какой-либо вывод о поведении программы;
  - покрытие программы не полно: динамические методы рассматривают только несколько возможных вариантов выполнения программы. Более того, множество значимых вариантов выполнения программы не может быть обнаружено;
  - эмуляция окружения, в котором анализируемая программа может проявить свои вредоносные свойства, сложна;

– существуют техники обнаружения выполнения программы в изолированном окружении. В этом случае программа имеет возможность изменить своё поведение с целью невыявления вредоносных свойств.

- *Гибридные методы.* В эту группу входят методы, использующие комбинацию статических и динамических методов анализа кода программы. Как правило, подобные методы основаны на построении некоторой модели программы путем статического анализа и дальнейшей проверки этой модели динамическим анализом.

3.3. Основные методы

3.3.1. Статические методы

Buttercup

Одним из примеров сигнатурных методов является **Buttercup** [76] - статический метод, ориентированный на поиск зоны адреса возврата во вредоносном объекте. Алгоритм использует значения адреса возврата из набора популярных эксплойтов, и проверяет значения, которые лежат в фиксированном диапазоне адресов относительно этих значений. Алгоритм рассматривает входной поток, разбитый на блоки по 32 бита. Значение каждого байта в блоке сравнивается с диапазонами адресов из сигнатур. Если значение байта попадает в один из промежутков, считается, объект S является вредоносным.

Вероятность ложных срабатываний метода оценивается значением 0.01%. Вычислительная сложность метода, без учета этапа дизассемблирования. оценивается значением $O(N)$, где N - длина исследуемого объекта S, вычисляемая в байтах. Рассматриваемым методом возможно обнаружение следующих классов вредоносного исполнимого кода: K_{PL}, K_{DATA}, K_{ALT-OP}, K_R, $K_{ALT-INS}$, K_{INJ}, K_{RET}.

Несмотря на то, что метод Buttercup потенциально способен обнаруживать многие классы вредоносного исполнимого кода, использование метода на реальных каналах проблематично. Это связано с тем, что метод использует сигнатуры адреса возврата, однако, исходя из анализа современных эксплойтов, статичный адрес практически не используется. Исключением являются некоторые виды встроенных систем, допускающие эксплуатирование ошибок работы с памятью с помощью статических адресов возврата. Тем не менее, метод может быть использован в качестве дополнительной проверки совместно с другими методами обнаружения вредоносного исполнимого кода, так как не требует больших временных и вычислительных затрат. Метод может быть использован на каналах с любым профилем трафика (с любой вероятность передачи исполнимого кода), а так же допускает применение к высокоскоростным каналам передачи данных в режиме реального времени.

Hamsa

Еще одним примером сигнатурного метода является **Hamsa** [70] - статический метод, строящий контекстно-зависимые сигнатуры на основе обучающей выборки из вредоносных объектов. Из обучающей выборки строится сигнатура $Sig_j = \{T_1, \ldots, T_k\}$, в данном методе представляющая из себя набор токенов $T_j = \{I_{j_1}, \ldots, I_{j_h}\}$, где I_{j_i} - инструкция. Таким образом, сигнатура, предлагаемая авторами метода - некоторое упорядоченное множество подмножеств инструкций, входящих в шеллкоды вредоносной выборки.

Для общего случая в [70] сформулирована следующая теорема:

Теорема 3. *Задача построения сигнатуры Sig для наперед заданного параметра $\rho < 1$, такого что значение ложных срабатываний метода для построенной сигнатуры не будет превышать значения этого параметра $FP(Sig) \leq \rho$, NP-трудная.*

В виду этого, авторами подразумеваются следующие допущения в задаче: пусть априорно заданы параметры $k^*, u(1), \ldots, u(k^*)$. Тогда, при гене-

рации сигнатуры токен T_i в нее добавляется в том и только в том случае, если значение ложных срабатываний уже сгенерированной на данный момент сигнатуры, включающей в себя заданный токен, не превосходит значения $u(i)$:

$$FP(Sig \bigcup \{t\}) \leq u(i).$$

Сгенерировав подобную сигнатуру, алгоритм проверяет ее соответствие заданному объекту S

Вероятность ложных срабатываний авторами метода оценивается значением 0.7%. Алгоритмическая сложность алгоритма оценивается формулой $O(T \times N)$, где N - длина проверяемого объекта S, а T - число токенов в сигнатуре объекта. Рассматриваемым методом возможно обнаружение следующих классов вредоносных объектов: K_{NOP-PL}, K_{NOP-B}, K_{NOP-MB}, $K_{NOP-4AL}$, K_{NOP-T}, K_{NOP-TO}, $K_{NOP-SAR}$, K_{PL}, K_{DATA}, K_{ALT-OP}, K_{RET_+}.

Несмотря на то, что Hamsa обеспечивает достаточно большое покрытие классов обнаруживаемых вредоносных объектов, метод не приспособлен к поиску полиморфных версий вредоносного исполнимого кода в виду специфики генерируемых сигнатур. Тем не менее, метод обладает пропускной способностью, приемлемой для применения его на реальных каналах передачи данных.

Polygraph

Последний из рассмотренных статических сигнатурных методов, строящих контекстно-зависимые сигнатуры представлен в работе [74], описывающей метод **Polygraph**. В качестве обучающей выборки, состоящей из объектов $S_1, \ldots, S_m$, алгоритм принимает различные версии одного и того же объекта S', к которому m раз была применена операция полиморфного изменения.

На основе обучающей выборки, Polygraph осуществляет автоматическое построение базы сигнатур, которая в дальнейшем используется для сравнения с анализируемыми образцами. Сигнатура представляет из себя

набор *токенов*, где токен - байтовая последовательность.

В качестве предварительного шага, метод производит извлечение токенов из обучающей выборки. На этом этапе происходит извлечение всех подстрок минимальной длины, встречающихся минимум K раз в выборке из n образцов. Для извлечения таких подстрок используется модификация алгоритма [59].

Из выделенного набора токенов, метод генерирует три следующих класса сигнатур::

- неупорядоченные сигнатуры Sig_C - сигнатуры, состоящие из неупорядоченного набора токенов. Набор исполнимых инструкций соответствует такой сигнатуре тогда и только того, когда он содержит каждый токен сигнатуры. Для генерации таких сигнатур, применяется алгоритм [59], из результатов работы которого выбираются токены, присутствующие в каждом образце обучающей выборки. Сложность генерирования такой сигнатуры линейна по размеру вредоносной выборки: $O(m)$, где m - размер вредоносной выборки;
- упорядоченные сигнатуры Sig_{SUB} - сигнатуры, состоящие из упорядоченного набора токенов. Набор исполнимых инструкций удовлетворяет подобной сигнатуре тогда и только тогда, когда он содержит упорядоченную подпоследовательность токенов сигнатуры. Для генерации подобны сигнатур необходимо выделение упорядоченного набора токенов, некоторая непустая подпоследовательность которых встречается в каждом из образцов вредоносной выборки. Для экстракции такого набора упорядоченных токенов используется модификация алгоритма Смита-Вотермана [92], обнаруживающий общие последовательности в молекулах белка. Сложность генерации таких сигнатур оценивается значением $O(sn^2)$, где s - количество образцов во вредоносной обучающей выборке, n - максимальный размер образца во вредоносной обучающей выборке;

- сигнатуры Байеса $Sig_B = \{(T_{B_1}, M_1), \ldots, (T_{B_r}, M_r)\}$. Для каждого токена таких сигнатур вычисляется информация о количестве появления токена в обучающей выборке, а так же каждому токену сопоставляется определенное пороговое значение. Такие сигнатуры дают возможность осуществлять сравнение вероятностного распределения токенов в сигнатуре и в анализируемом образце. Для генерации таких сигнатур применяется наивный байесовский классификатор [30].

Зададим несколько следующих предикатов:

1. $P_C(S) = (Sig \in S)$ - предикат проверяет вхождение всех токенов сигнатуры в объект S.
2. $P_{SUB}(S) = (\forall i, j, m, n, k, t : T_{SUB_i} = \{I_m, \ldots, I_n\}, T_{SUB_j} = \{I_k, \ldots, I_t\} : i < j \Rightarrow m < n)$ - предикат проверяет упорядоченность вхождения токенов в объект.
3. $P_B(S) = (\forall i : |T_{B_i}| \geq M_i)$ - предикат проверяет превышение токеном заданного порога.

Тогда отнесение алгоритмом Polygraph объекта S к одному из вредоносных классов можно задать как выполнение следующей предикатной формулы:

$$P_C(S) \vee P_C(S) \& P_{SUB}(S) \vee P_B(S)$$

На практике вредоносные обучающие выборки редко содержат образцы, содержащие только один тип вредоносного исполнимого кода. В виду этого, авторы модифицировали Polygraph таким образом, чтобы средство могло генерировать не единственную сигнатуру, а набор сигнатур для разных типов вредоносного исполнимого кода. Авторами доказывается утверждение, что алгоритм генерации байесовских сигнатур может использоваться в этом случае без модификаций. Для двух других типов сигнатур необходим предварительный шаг - кластеризация вредоносной обучающей выборки. Далее, средством генерируются сигнатуры для каждого отдельно взятого кластера.

Вероятность ложных срабатываний авторами метода оценивается значением 0.2%. В зависимости от применение или не применения кластерного анализа, алгоритмическая сложность метода оценивается следующими значениями:

- Без кластеризации: $O(N)$, где N - длина проверяемого объекта S;
- С кластеризацией: $O(N + C^2)$, где N - длина проверяемого объекта S, а C - количество кластеров.

При этом, алгоритмическая сложность обучения метода определяется значением $O(M^2 \times L)$, где M - длина вредоносной обучающей выборки, а L - длина легитимной обучающей выборки. Алгоритмическая сложность алгоритма оценивается значением $O(T \times N)$, где N - длина проверяемого объекта S, а T - число токенов в сигнатуре объекта. Рассматриваемым методом возможно обнаружение следующих классов вредоносных объектов: K_{NOP-PL}, K_{NOP-B}, K_{NOP-MB}, $K_{NOP-4AL}$, K_{NOP-T}, K_{NOP-TO}, K_{PL}, K_{DATA}, K_{ALT-OP}, K_{INJ}.

Метод Polygraph, так же, как и Hamsa, основывается на построение контекстно-зависимых сигнатур. Различные виды сигнатур метода Polyghaph обеспечивают методу большую гибкость, хотя в общем случае, полиморфные версии вредоносного исполнимого кода методом не обнаруживаются. Все три класса сигнатур метода имеют и преимущества, и недостатки. Упорядоченные сигнатуры более специфичны, чем эквивалентные им неупорядоченные сигнатуры. Кроме того, некоторые варианты вредоносного исполнимого кода могут содержать некоторые шаблоны - константные значения или зарезервированные ключевые слова, которые могут появляться в любом порядке. В этом случае неупорядоченные сигнатуры более предпочтительны. Сигнатуры Байеса генерируются гораздо быстрее других сигнатур и более эффективны в тех случаях, когда шаблоны возникают в образцах вредоносного исполнимого кода не часто. Авторы метода рекомендуют использовать все три вида сигнатур одновременно, однако это влечет

за собой существенные накладные расходы. К примеру, Polygraph в лучшем случае в 64 раза медленнее метода Hamsa, а в худшем это значение достигает 361 раза [70]. Пропускная способность алгоритма накладывает ограничения на его использовании на реальных каналах передачи данных.

Структурный анализ

Метод структурного анализа, предложенный в работе [68], представляет из себя метод сигнатурного анализа, генерирующего сигнатуры *поведения* программы. В процессе обучения на выборке, состоящей из вредоносных объектов $S_1, \ldots, S_m$, метод строит базу сигнатур поведения программы. Проверяемый объект S считается вредоносным, если сигнатура его поведения содержится в такой базе. Извлечение сигнатуры поведения из программы осуществляется следующим образом:

- с помощью анализа графа потока управления программы определяется её структура;
- идентифицируются объекты программы при помощи раскрашивания графа потока управления;
- для каждой сигнатуры и построенной структуры программы анализируется, являются ли они полиморфной модификацией друг друга - то есть описывают ли они поведение одной и той же программы, к которой была применена некоторая полиморфная обфускация.

Тем не менее, сравнение графов потока управления неэффективно в виду неустойчивости такого метода к любым модификациям (например, добавление в программу линейных блоков, состоящих из мусорных инструкций, не влияющих на ход выполнения программы, влечет за собой неравенство двух графов потока управления). Авторами статьи предлагается следующая модернизация этого метода: осуществляется идентификация подграфов графа потока управления, содержащих k вершин. Идентификация подграфа осуществляется следующим образом:

- Подграф переводится в каноническую форму [24] с помощью библиотеки Nauty [73], [72].
- Строится матрица смежности. В этой матрице строки и столбцы отвечают вершинам подграфа, а позиция (i, j) может принимать значения 0 или 1 в зависимости от того, присутствует ли в подграфе ребро из v_i в v_j или нет;
- Строки матрицы смежности конкатенируются и рассматриваются как двоичный идентификатор. Пример генерации идентификатора из графа, содержащего 4 вершины, приведен на рисунке 3.1 [68];

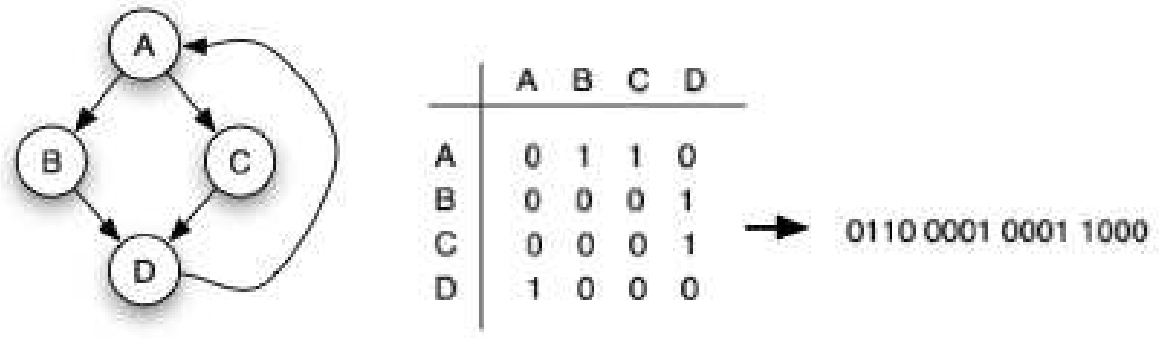


Рис. 3.1: Пример генерации идентификатора подграфа.

- Перед конкатенацией строк добавляется информация о раскраске графа. Раскраска графов осуществляется путем анализа инструкции в соответствующей вершине. Все инструкции, таким образом, разделяются на 14 классов:
  1. Инструкции передачи данных (`mov`);
  2. Арифметические инструкции, включая `shift` и `rotate`;
  3. Логические инструкции, включая битовые/байтовые операции;
  4. Тестовые инструкции (`test, compare`);
  5. Инструкции работы со стеком (`push, pop`);
  6. Инструкции условного перехода;
  7. Вызовы функций;

8. Инструкции работы со строками;
9. Инструкции работы с флагами;
10. `lea`;
11. Операции с плавающей точкой;
12. Системные вызовы и прерывания;
13. Инструкции безусловного перехода;
14. `halt`.

Определение 1. *Два подграфа считаются родственными, если они изоморфны с точностью до раскраски вершин.*

Определение 2. *Два графа потока управления считаются родственными, если они содержат родственные k-подграфы (подграф, содержащий k вершин).*

Считается, что если $CFG(S)$ родственен какому-либо графу потока управления вредоносного объекта, то сам объект S вредоносен.

Вероятность ложных срабатываний авторами метода оценивается значением 0.5%. Алгоритмическая сложность алгоритма оценивается значением $O(N)$, где N - длина проверяемого объекта S. Рассматриваемым методом возможно обнаружение следующих классов вредоносных объектов: K_{PL}, K_{DATA}, K_{ALT-OP}, $K_{ALT-INS}$, K_{INJ}, K_{RET}, K_{RET_+}, K_{MET}.

Метод, представленный в работе [68], в отличии от Hamsa и Polygraph, позволяет обнаруживать некоторые виды обфусцированных образцов вредоносных объектов и в отдельных случаях - метаморфные образцы вредоносных объектов, так как ориентирован на построение сигнатур поведения программы. Несмотря на то, что алгоритмическая сложность всех трех методов сравнима, метод структурного анализа медленнее двух остальных алгоритмов. Анализ трафика, в виду временных затрат на работу алгоритма, возможен только в режиме оф-лайн. Еще одним недостатком метода является невозможность обнаружения небольших по размеру образцов вре-

доносного исполнимого кода. В прототипе алгоритма число инструкций обнаруживаемого вредоносного исполнимого кода должно превышать 10 инструкций.

Stride

Алгоритм **Stride** [18] принадлежит классу методов, обнаруживающих NOP-след. В заданной последовательности инструкций, алгоритм ищет NOP-след длины n, начиная рассмотрение в каждой из инструкций. Формально STRIDE можно описать следующим образом: для очередной последовательности байтов входного потока длины n формируется объект S путем дизассемблирования, начиная с каждой позиции в последовательности с 0 по $n-1$ или с каждой четвёртой позиции. Считается, что во входе найдена подпоследовательность из n инструкций, являющая NOP-следом, если она дизассемблируется, начиная с каждого байта (либо с каждого 4-го байта), и в каждой подпоследовательности объекта S не встречается привилегированная инструкция, либо же до нее встретилась инструкция перехода.

Доля ложных срабатываний метода оценивается авторами значением 0.0027%. Тем не менее, существует вероятность того, что NOP-эквивалентная последовательность байт может встретиться и в легитимном трафике. Например, такая последовательность может встретиться как в части ELF-файлах, ACSCII тексте, мультимедийных файлах и так далее. Авторы метода не учитывают этот факт при вычислении доли ошибки второго рода, таким образом реальное значение доли ошибок второго рода может существенно отличаться от приведенного. Реальное значение вероятности ложных срабатываний метода может быть существенно выше. Алгоритмическая сложность алгоритма оценивается значением $O(N \times l^2)$, где N - длина проверяемого объекта S, а l - длина NOP-следа. Рассматриваемым методом возможно обнаружение следующих классов вредоносных объектов: K_{NOP-PL}, K_{NOP-B}, K_{NOP-MB}, $K_{NOP-4AL}$, K_{NOP-T}, K_{NOP-TO}. Метод, в виду невысокой алгоритмической сложности, может быть использован на

реальных каналах передачи данных.

Racewalk

Алгоритм **Racewalk** [54] улучшает показатели работы алгоритма Stride за счет введения кэша декодированных инструкций, техник отбрасывания инструкций, не являющихся NOP-следом (например, если в некоторой позиции h была встречена привилегированная или некорректная инструкция, очевидно, что объект S, сформированный со смещения $j = h\%4$ не будет причислен ни к одному из классов $K_{NOP_1}, \ldots, K_{NOP_7}$), а также за счет оптимизации процесса дизассемблирования путем построения префиксного дерева инструкций.

Доля ложных срабатываний метода оценивается значением 0.0058%. Алгоритмическая сложность алгоритма оценивается значением $O(N \times l)$, где N - длина проверяемого объекта S, l - длина NOP-следа. Рассматриваемым методом возможно обнаружение следующих классов вредоносных объектов: K_{NOP-PL}, K_{NOP-B}, K_{NOP-MB}, $K_{NOP-4AL}$, K_{NOP-T}, K_{NOP-TO}.

Styx

Styx [32] - метод статического анализа, основанный на исследовании графа потока управления. Объект S считается вредоносным, если в специальным образом обработанном графе потока управления содержатся циклы, что свидетельствует о полиморфной природе объекта. Для такого объекта генерируется сигнатура с целью дальнейшего ее использования.

Алгоритм для заданного объекта S строит граф потока управления (CFG). В качестве вершин графа выступают блоки цепочек инструкций, исполняемых последовательно и не содержащих переходов, ребра являются соответствующими переходами между блоками. Все линейные блоки в графе разделяются на три класса:

1. корректные линейные блоки – класс блоков, последняя инструкция которых содержит переход на корректную инструкцию;
2. некорректные линейные блоки – класс блоков, последняя инструкция

которых содержит переход на некорректную инструкцию;

3. неизвестные линейные блоки – класс блоков, для которых корректность перехода последней инструкции неочевидна.

Из GFG строится граф потока управления, из которого исключены все некорректные блоки, а так же блоки, на которые они имеют переходы. Часть блоков исключается так же с использованием техники анализа потока данных, описанной в работе [103]. Из модифицированного CFG строится множество всевозможных путей, каждый из которых исследуется независимо на наличие циклов.

Доля ложных срабатываний метода оценивается значением 0%. Стоит отметить, что подобное значение достижимо только на образцах вредоносного исполнимого кода обнаруживаемых методом классов. Алгоритмическая сложность метода оценивается значением $O(N)$, где N - длина проверяемого объекта S. Рассматриваемым методом возможно обнаружение следующих классов вредоносных объектов: $K_{NOP-SAR}$, K_{SU}, K_{SC}.

Метод Styx в состоянии обнаруживать самошифрующиеся и самораспаковывающиеся образцы вредоносного исполнимого кода, однако в среднем случае работает медленнее аналогичных методов динамического анализа. Пропускная способность метода существенно сокращает его сферу применимости. Тем не менее, метод может использоваться как дополнение к другим алгоритмам обнаружения вредоносного исполнимого кода, увеличивая покрытие обнаруживаемых классов.

SigFree

В отличие алгоритма Styx, метод **SigFree** [102] использует для статического анализа не граф потока управления, в вершинах которого содержатся блоки инструкций, а граф потока инструкций IFG объекта S. Объект считается вредоносным, если его поведение соответствует поведению реальной программы, а не случайного набора инструкций. Подобная эвристика сильно ограничивает применимость метода на каналах, где профиль тра-

фика допускает передачу исполнимых программ.

Определение 3. *Граф потока инструкций (IFG) - ориентированный граф $G = (V, E)$, в котором каждой вершине v сопоставлена инструкция, а каждому ребру $e = (v_i, v_j) \in E$ сопоставлен возможный переход управления от инструкции v_i к инструкции v_j.*

Анализ основывается на предположении, что легитимный объект S, составленный из инструкций, встретившихся во входном потоке, не может быть фрагментом реальной программы. Реальной программе приписываются два важных свойства:

1. Программа имеет специфичные характеристики, индуцированные операционной системой, на которой она запущена. К примеру, это системные вызовы или обращение к библиотеке ядра.
2. Программа имеет определенное число *значимых инструкций* - инструкций, влияющих на поток управления программы.

Относительно перечисленных свойств, в методе предусмотрены две схемы анализа графа IFG. В первой схеме, на основе обучающей выборки строится множество $\{TEMPL\}$ шаблонов *вызовов* инструкций, после чего объект S проверяется на соответствие этим шаблонам. Вторая схема анализа графа IFG основана на анализе потока данных. В этой схеме каждой переменной может быть сопоставлено значение из множества состояний $Q = \{U, D, R, DD, UR, DU\}$, где шесть возможных состояний переменной определены следующим образом:

1. U (undefined) - переменная не определена;
2. D (defined) - переменная определена, но не была использована;
3. R (refered) - переменная определена и была использована;
4. UR (undefined-refered) - переменная не определена, но была использована. Примером может служить следующий набор инструкций:

   0: `mov ax, 5`

1: `mov ax, bx`

2: `sub ax, 0`

Несмотря на то, что на шаге 0 переменная, ассоциированная с регистром `ax` была определена, на шаге 1 ей присваивается неопределенное значение, содержащеесяв регистре `bx`. Таким образом, на шаге 2, переменная будет иметь состояние UR.

5. DD (defined-defined) - определенная переменная была переопределена без использования. Примером такого состояния может слушить переменная на шаге 2 следующего набора инструкций:

   0: `mov ax, 5`

   1: `mov bx, 2`

   2: `mov ax, bx`

6. DU (defined-undefined) - определенной переменной было присвоено неопределенное значение без ее использования. Примером может служить состояние переменной, ассоциированной с регистром `ax` на шаге 1:

   0: `mov ax, 5`

   1: `mov ax, bx`

Состояния $F = \{DD, UR, DU\}$ считаются аномальными. Для объекта S строится диаграмма состояний переменных - автомат $DSV = (Q, \Sigma, \delta, q_0, F)$, Σ - алфавит, состоящий из инструкций объекта S, а $q_0 = U$ - начальное состояние. Если при разборе DSV автоматом объекта S был осуществлен переход на заключительное (аномальное состояние), то считается, что инструкция, по которой был осуществлен переход, не является значимой. Все незначимые инструкции исключаются из объекта S, в результате чего образуется объект $S' \subset S$, состоящий из значимых инструкций. Вычисляя мощность объекта S', алгоритм делает вывод о вредоносности объекта S.

Доля ложных срабатываний метода оценивается авторами значением 0% [102] . Тем не менее, алгоритм подразумевает анализ такого трафика, профиль которого не допускает передачу *любого* исполнимого кода. Для каналов, допускающих передачу исполнимого кода, метод будет характеризоваться значительным числом ложных срабатываний. Алгоритмическая сложность метода оценивается значением $O(N)$, где N - длина проверяемого объекта S. Рассматриваемым методом возможно обнаружение следующих классов вредоносных объектов: K_{NOP-PL}, K_{NOP-B}, K_{NOP-MB}, $K_{NOP-4AL}$, K_{NOP-T}, K_{NOP-TO}, $K_{NOP-SAR}$, K_{DATA}, K_{ALT-OP}, K_{INJ}, $K_{ALT-INS}$, K_R, K_{RET}, K_{RET_+}, K_{MET}.

STILL

Метод **STILL** [101] является усовершенствованием метода $SigFree$. Метод включает в себя статичный "анализ пятен" и инициализационный анализ для обнаружения самомодифицирующегося кода. Метод основан на предположении, что самомодифицирующийся код и код, использующий косвенные переходы, должен вычислить абсолютный адрес полезной нагрузки шеллкода. В виду этого, в объекте S ищется подмножество инструкций S', которое получает абсолютный адрес полезной нагрузки во время выполнения. Переменная, в которую записывается абсолютный адрес помечается как "зараженная". Зараженная переменная отслеживается посредством статического анализа с целью выявления ее использования такими способами, которые могут указывать на наличие самомодифицирующегося кода и кода, использующего косвенную адресацию. Переменная может заразить другие посредством инструкций передачи данных (`push, pop, move`), и инструкций, выполняющих арифметические и логические операции над данными (`add, sub, xor`).

Для уменьшения количества ложных срабатываний метод использует анализ инициализации. Анализ строится на предположении, что операнды самомодифицирующегося кода и кода, использующего косвенные перехо-

ды, должны быть инициализированы. Если это не так, объект s считается легитимным.

Доля ложных срабатываний метода оценивается значением 0%. Алгоритмическая сложность метода оценивается значением $O(N)$, где N - длина проверяемого объекта S. Рассматриваемым методом возможно обнаружение следующих классов вредоносных объектов: K_{NOP-PL}, K_{NOP-B}, K_{NOP-MB}, $K_{NOP-4AL}$, K_{NOP-T}, K_{NOP-TO}, $K_{NOP-SAR}$, K_{DATA}, K_{ALT-OP}, K_{INJ}, $K_{ALT-INS}$, K_{RET}, K_{RET_+}, K_{MET}, K_{SU}, K_{SC}.

Методы SigFree и STILL дополняют друг друга, обеспечивая практически полное покрытие классов вредоносного исполнимого кода. Кроме того, применение методов возможно в режиме реального времени на высокоскоростных каналах. Однако ошибки второго рода обоих методов имеют нулевое значение лишь для такого профиля трафика, в котором не подразумевается наличия какого бы то ни было исполнимого кода. Для профиля трафика, допускающего передачу исполнимого кода, ошибка второго рода этих методов очень велика. Этот факт ограничивает применимость SigFree и STILL.

SAMD

В работе [33] применен подход сигнатурного анализа поведения программы. Метод посредством обучения на выборке, состоящей из вредоносных объектов, строит множество сигнатур-шаблонов их поведения. Проверяемый объект S считается вредоносным, если его поведение соответствует хотя бы одному шаблону из этого множества.

Авторами [33] приводится и доказывается следующее утверждение:

Теорема 4. *Проблема определения соответствия программы какому-либо поведению (TMP-проблема) в общем случае неразрешима.*

В качестве вывода из доказанного в работе утверждения, авторы метода указывают на то, что их метод не может иметь полное покрытие классов вредоносных объектов, определяя вредоносность объекта для ограниченно-

го числа модификаций исполнимых инструкций. По обучающей выборке алгоритмом строится множество $\{T\}$ шаблонов вредоносного поведения программы. Считается, что объект S удовлетворяет шаблону, если выполнены следующие условия:

- значения в адресах, которые были изменены в процессе выполнения инструкций объекта S, одинаковы после выполнения шаблона с соответствующим контекстом;
- последовательность системных вызовов шаблона является подпоследовательностью системных вызовов набора проверяемых инструкций;
- если в процессе выполнения шаблона счетчик команд указывает на область памяти, значение которой изменялось в процессе выполнения, в случае выполнения набора проверяемых инструкций ситуация должна быть аналогичной.

Для проверки соответствия объекта S шаблону поведения, метод проверяет соответствие вершин шаблона вершинам объекта S, а так же осуществляет построение "def-use" путей и их проверку. *Сопоставление вершин шаблона вершинам программы* осуществляется с помощью построения графа потока управления CFG, учитывая следующие правила:

- переменная шаблона может быть ассоциирована с любой инструкцией $I_j \in S$, являющейся программным выражением, за исключением инструкций присвоения;
- символьная константа шаблона может быть ассоциирована только с константой в S;
- функция над памятью может быть ассоциировано только с функцией над памятью;
- оператор шаблона может быть ассоциирован только с таким же оператором в S;
- внешние функции могут быть ассоциированы с вызовом этих же функций.

Выделение def-use путей и их проверка. Под Def-use путем понимается такая цепочка вершин шаблона (или CFG - графа объекта S), в первой вершине которой определяется переменная, а в последней эта переменная используется. Для каждого def-use пути шаблона выделяется фрагмент объекта S, отвечающий этому пути, и осуществляется проверка, сохраняется ли переменная в инвариантном значении или нет - происходило ли изменение переменной в шаблоне и объекте одинаково. Для решения проблемы сохранения переменной используются следующие процедуры:

- определение NOP участка - осуществляется путем простого сопоставления сигнатур;
- вторая процедура ориентирована на нахождение участков, в которых инвариация переменной не сохраняется. Фрагмент программы выполняется с произвольного начального состояния;
- доказательство соответствующей теоремы с использованием simplify [40] или UCLID [28]. Подробнее методы описаны в соответствующей литературе.

Доля ложных срабатываний метода оценивается значением 0%. Тем не менее, алгоритм подразумевает анализ такого трафика, профиль которого не допускает передачу *любого* исполнимого кода. Для каналов, допускающих передачу исполнимого кода, метод будет характеризоваться значительным числом ложных срабатываний. Алгоритмическая сложность метода оценивается значением $O(N)$, где N - длина проверяемого объекта S. Рассматриваемым методом возможно обнаружение следующих классов вредоносных объектов: K_{PL}, K_{DATA}, K_{ALT-OP}, K_R, K_{RET}, K_{RET_+}.

Так же, как и Styx, рассматриваемый метод основан на анализе графа потока управления. Метод характеризуется низкой скоростью работы, что не позволять применять его в режиме реального времени даже на каналах с невысокой пропускной способностью. Недостатком метода так же является тот факт, что могут быть обнаружены только те образцы вредоносного

исполнимого кода, которые содержат тот же порядок обновлений в памяти, что и образцы вредоносного исполнимого кода из обучающей выборки. Тем не менее, метод может использоваться совместно с другими средствами обнаружения вредоносных исполнимых кодов в качестве дополнительной проверки.

APE

APE [99] — механизм обнаружения NOP-следа, основанный на поиске достаточно длинных последовательностей корректных инструкций, чьи операнды в памяти лежат в адресном пространстве защищаемого процесса. Для уменьшения вычислительных затрат, выбирается небольшое количество позиций в исследуемых данных, начиная с которых осуществляется *абстрактное выполнение* - проверка инструкций объекта S на корректность и допустимость операндов в памяти. Число корректных инструкций, декодируемых начиная с каждой выбранной позиции обозначается MEL (Maximum Executable Length). При анализе команды условного перехода методом APE исследуются обе ветви и выбирается наибольшая величина MEL. Если цель перехода невозможно вычислить статически, команда перехода объявляется некорректной. Считается, что обнаружен NOP-след, если величина MEL достигла определенного порога Thr.

Доля ложных срабатываний метода оценивается значением 0%. Алгоритмическая сложность метода оценивается значением $O(N \times 2^l)$, где N - длина проверяемого объекта S, l - длина NOP-следа. Рассматриваемым методом возможно обнаружение следующих классов вредоносных объектов: K_{NOP-PL}, K_{NOP-B}, K_{NOP-MB}, $K_{NOP-4AL}$, K_{NOP-T}.

Аналогичные APE методы Stride и Racewalk, обнаруживающие NOP-след в байтовом потоке данных, характеризуются существенно большей скоростью работы. В виду этого, применение метода APE на реальных каналах передачи данных не оправданно.

3.3.2. Динамические методы

Emulation

Одним из примеров динамических методов обнаружения вредоносного объекта является метод **Emulation** [78]. Алгоритм анализирует цепочки инструкций, полученных в результате выполнения исследуемого объекта S в изолированном виртуальном окружении. Выполнение осуществляется с каждого байтового смещения объекта S. Таким образом, из исходного объекта S генерируется множество объектов $\{S'_i \,|S'_i \subset S\}$. Если хотя бы один из объектов S_i удовлетворяет определенным эвристикам, объект S считается вредоносным. К таким эвристикам относятся следующие:

- Исполнение объектом S'_i какой-либо из форм GetPC кода - кода, определяющего свое расположение в адресном пространстве исполнимого процесса. Эвристика связана с необходимостью вредоносного объекта вычислить исполнимый адрес;
- Количеством доступа к памяти превышает определенный порог. Эвристика связана с характерной для самошифрующихся и самомодифицирующихся шеллкодов особенностью цикличного дешифрования/модификации небольшого участка памяти.

Метод считает объект S'_i легитимным, если в ходе его исполнения была встречена некорректная или привилегированная инструкция.

Доля ложных срабатываний метода оценивается значением 0.004%. Алгоритмическая сложность метода оценивается формулой $O(N^2)$, где N - длина проверяемого объекта S, вычисляемая в байтах. Рассматриваемым методом возможно обнаружение следующих классов вредоносных объектов: $K_{NOP-SAR}$, K_{MET}, K_{SU}, K_{SC}.

Существенным плюсом рассматриваемого метода является его устойчивость к техникам обфускации, используемые для уклонения от обнаружения статическим анализом. В то же время, метод имеет ограниченную

область применения, так как обнаруживает только классы, содержащие подобные обфускации - метод обнаруживает только образцы вредоносного исполнимого кода, заменяющего свое тело в процессе выполнения. К таким классам относятся образцы вредоносного исполнимого кода, использующие техники самораспаковки и самошифрования, широко применимые в современных эксплойтах.

NSC emulation

Метод **NSC emulation** [77] является расширением метода [78], сфокусированном на обнаружении NSC (non-self-contained) образцов вредоносного исполнимого кода. Такой код должен удовлетворять двум следующим условиям:

1. в наборе вредоносных исполнимых инструкций не содержится какой-либо формы GetPC кода;
2. набор вредоносных исполнимых инструкций не выполняет операций чтения из своего собственного адресного пространства во время исполнения.

Для исследуемого объекта S производится эмуляция его исполнения, начиная со всех возможных позиций. Объект относится к классу K_{NSC}, если выполнена следующая эвристика. Пусть *уникальная запись в память* - запись в память по уникальным адресам, а wx - *инструкция* - инструкция, которая была записана в память в процессе исполнения какой-либо инструкции из объекта S. Пусть W и X - пороговые значения для уникальных записей и wx-инструкций, соответственно. Тогда объект относится к классу K_{NSC}, если после его выполнения эмулятором было произведено как минимум W уникальных записей и выполнено как минимум X wx-инструкций.

Доля ложных срабатываний метода оценивается значением 0%. Стоить отметить, что значение вычислено только для единственного обнаруживаемого класса вредоносного исполнимого кода - K_{NSC}. Алгоритмическая

сложность метода оценивается значением $O(N^2)$, где N - длина проверяемого объекта S.

Так же, как и метод Emulation, NSC-Emulation устойчив к антистатическим методам уклонения от обнаружения. Тем не менее, метод способен обнаруживать единственный класс вредоносных исполнимых инструкций, который практически не встречается на практике. В виду этого, область применимости метода не ясна.

IGPSA

IGPSA [100] - метод, так же использующий эмуляцию. Метод основывается на эмуляции исполнения инструкций, информация о которых обрабатывается конечным автоматом. Во время цикла эмуляции объекта S алгоритм генерирует так называемую последовательность трансформированных шаблонов W, которая состоит из элементов множества $\{P_1, \ldots, P_5\}$. Все инструкции объекта S относятся к одной из пяти категорий, представленные шаблонами P_i этого множества, соответственно:

1. если инструкция производит запись PC в конкретную область памяти, она относится к шаблону P_1;
2. если инструкция извлекает PC из памяти - к P_2;
3. если инструкция производит чтение из памяти, относящейся к адресному пространству кода, - к P_3;
4. если пишет данные в память по адресу PC - к P_4;
5. все остальные инструкции относятся к шаблону P_5.

На основе этого методом строится автомат $IGPA = (Q, \Sigma, \delta, q_0, F)$, где Q - множество состояний, соответствующих типичному поведению полиморфного шеллкода (GetPC коде и цикл дешифровки); $\Sigma = \{P_1, \ldots, P_5\}$ - алфавит; δ - функций перехода; q_0 - начальное состояние; F - множество конечных состояний. Пример автомата для набора инструкций, приведенных на рисунке 3.2 [100], представлен на рисунке 3.3 [100].

00:	EB10	jmp12	P_5
02:	5A	pop edx	P_2
03:	4A	dec edx	P_5
04:	33C9	xor ecx, ecx	P_5
06:	66B97D01	mov cx, 017D	P_5
0A:	80340A99	xor byte ptr [edx+ecx], 99	P_3 P_4
0E:	E2FA	loop 0A	P_5
10:	EB05	jmp 17	P_5
12:	E8EBFFFFFF	call 02	P_1
17:			
...	<encrypted payload>		

Рис. 3.2: Пример генерации идентификатора подграфа.

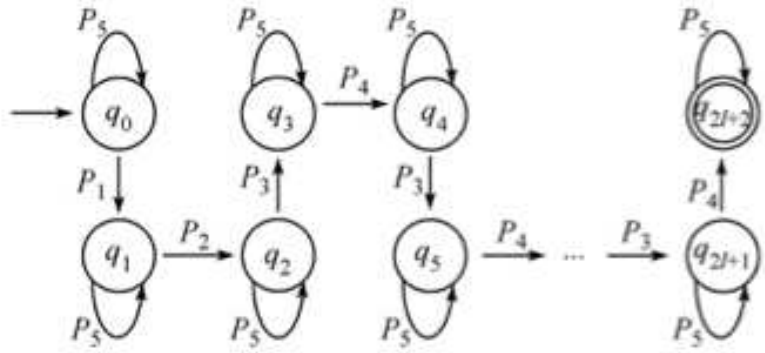


Рис. 3.3: Пример генерации идентификатора подграфа.

Таким образом, объект S считается вредоносным, если последовательность трансформированных шаблонов W разбирается автоматом $IGPA$.

Доля ложных срабатываний метода оценивается значением 0%. Алгоритмическая сложность метода оценивается значением $O(N^2)$, где N - длина проверяемого объекта S. Рассматриваемым методом возможно обнаружение следующих классов вредоносных объектов: $K_{NOP-SAR}$, K_{MET}, K_{SU}, K_{SC}.

Sandboxing

К динамическим методам обнаружения шеллкодов относится так же метод **Sandboxing** [20]. Метод использует плагин FNORD системы SNORT [14] для обнаружения NOP-инструкций в исследуемом объекте S. Обнаружение NOP-инструкций служит активатором для метода. В этом случае

осуществляется исполнение соответствующей полезной нагрузки в так называемой "песочнице"– изолированной среде с тем же самым программно-аппаратным окружением, которое установлено на машине-жертве. При исполнении в песочнице, посредством системного вызова `ptrace`, строится множество системных вызовов исследуемого объекта S. Тот факт, что множество системных вызовов объекта не пусто, интерпретируется как признак обнаружения вредоносного кода.

Доля ложных срабатываний метода оценивается значением 0%. Тем не менее, авторами не проводилось тестирование на легитимных исполнимых файлах. Таким образом, значение ложных срабатываний, при применении метода на каналах, допускающих передачу исполнимого кода, будет ненулевым. Алгоритмическая сложность метода оценивается значением $O(N)$, где N - длина проверяемого объекта S. Рассматриваемым методом возможно обнаружение следующих классов вредоносных объектов: $K_{NOP-SAR}$, K_{DATA}, K_{ALT-OP}, K_{INJ}, $K_{ALT-INS}$, K_R, K_{MET}, K_{SU}, K_{SC}. Существенным недостатком метода является его зависимость от среды исполнения.

3.3.3. Гибридные методы

HDD

Одним из примеров гибридных методов является **метод обнаружения самошифрующегося кода** [107] (HDD), предложенный К. Жангом. Статическая часть метода включает в себя двунаправленный анализ потока данных [64] (или объекта S), в результате которого выделяются подозрительные подмножества инструкций, входящих в объект S. Наличие поведения, свойственного для самошифрующегося кода проверяется путем эмуляции подозрительных подмножеств.

Первоначально, на этапе статического анализа, метод выполняет прямой рекурсивный анализ потока инструкций, начиная с подозрительной.

Подозрительной считается такая инструкция, которая может демонстрировать поведение $GetPC$ кода (например, `call, fnstenv` и др.). Метод запускает обратный анализ, если в результате прямого была встречена одна из следующих инструкций-активаторов:

- инструкция, которая осуществляет запись в память;
- инструкция условного перехода с косвенной адресацией.

Путем обратного анализа ищется *def-use* путь, заканчивающийся инструкцией-активатором. Такие цепочки $S_i \subset \{two\_way\_analysis(S)\}$ инструкций проверяются эмуляцией на предмет наличия циклов и записи в память, входящую в адресное пространство кода, что является указателем на самомодификацию.

Доля ложных срабатываний метода оценивается значением 0.0126%. Алгоритмическая сложность метода оценивается значением $O(N+K^2 \times T^2)$, где N - длина проверяемого объекта S, K - количество подозрительных цепочек, T - длина максимальной подозрительной цепочки. Рассматриваемым методом возможно обнаружение следующих классов вредоносных объектов: $K_{NOP-SAR}$, K_{MET}, K_{SU}, K_{SC}.

Метод может быть использован на каналах, характеризующихся сравнительно небольшой пропускной способностью в режиме реального времени. Существенным достоинством метода является его возможность обнаруживать метаморфные образцы вредоносного исполнимого кода, наряду с другими классами, использующими антистатические техники обфускации. Однако, тестирование метода на самораспаковывающихся программах, не носящих вредоносный характер, не проводилось. Этот факт может потенциально изменить значение доли ложных срабатываний, указанное авторами.

PolyUnpack

PolyUnpack [85]- гибридный метод, основанный на построении модели объекта S путем статического анализа, и проверки этой модели путем эмуляции. Объект считается *легитимным*, если он не генерирует данных,

подлежащих дальнейшему исполнению. Иначе объект считается самораспаковывающейся программой. На этапе статического анализа объект S разделяется на блоки кода и блоки данных. Из блоков кода, разделенных блоками данных, формируются последовательности инструкций $Sec_0, \ldots, Sec_n$. Эти последовательности формируют модель программы. На этапе динамического анализа, программа исполняется в изолированном окружении, останавливаясь после каждой выполненной инструкции. Статической модели отсылается запрос, содержащий эту инструкцию. Если инструкция соответствует статической модели, выполнение программы продолжается (при этом учитывается поддержание памяти в согласованном состоянии: последовательность инструкций между блоками данных должна корректно переводиться в последовательность инструкций в статической модели. Если найдена инструкция, не соответствующая статической модели, считается, что найден самораспаковывающийся код.

Доля ложных срабатываний метода оценивается значением 0% [85]. Алгоритмическая сложность метода оценивается значением $O(N)$, где N - длина проверяемого объекта S. Несмотря на невысокую алгоритмическую сложность, пропускная способность метода низка в виду необходимости проверки соответсвия *каждой* из выполняемых инструкций статической модели. Рассматриваемым методом возможно обнаружение следующих классов вредоносных объектов: $K_{NOP-SAR}$, K_{MET}, K_{SU}, K_{SC}, K_{NSC}.

Как было отмечено, скорость работы метода существенно ниже аналогичных ему, что объясняется большими временными затратами на построение модели программы и затратами на передачу сообщений "запрос-ответ"между выполняемой программой и ее моделью. Кроме того, с увеличением размера анализируемой программы, значение скорости работы алгоритма будет резко снижаться. Тем не менее, метод характеризуется стопроцентной точностью обнаружения и нулевой долей ложных срабатываний, что дает возможность использовать метод в качестве дополнительного

анализатора совместно с другими средствами обнаружения вредоносного исполнимого кода.

3.4. Результаты обзора

В данном разделе приведены результаты анализа и сравнения вышеперечисленных методов.

Доля ошибок первого рода методов, как следует из проведенных авторами соответствующих работ вычислений, оценивается значением 0% для всех перечисленных методов. Тем не менее, сравнение методов по этому критерию крайне затруднительно в виду того, что в каждом отдельном случае тестирование проводилось на совершенно разных наборах данных. Используемые авторами методов наборы данных, а так же значения ошибок первого и второго рода, перечислены в таблицах 3.1, 3.2 и 3.3. Ошибки первого и второго рода рассчитывались для тех классов вредоносных объектов, которые методы покрывают. Стоить заметить, что несмотря на относительно невысокие показатели процента ошибок второго рода, количество ложных срабатываний методов достигает больших значений на реальных каналах, характеризующихся большим объемом передаваемых данных.

Следующая таблица 3.4 резюмирует алгоритмическую вычислительную сложность предоставленных методов.

Таблица 3.5, демонстрируют результаты сравнения для каждого из классов методов по полноте покрытия классов K_j вредоносных объектов множества M.

Из приведенного обзора существующих решений по обнаружения вредоносного исполнимого кода, можно сделать следующие выводы:

1. *Методы, обладающие невысокой вычислительной сложностью, характеризуются большим процентом ложных срабатываний.* Невысокая алгоритмическая сложность, и, в ряде случаев, высокая про-

Method	FP, %	FN, %	Тестовые данные
Buttercup	0.01	0	Файлы TCPdump, собранные IDS университета MIT
Hamsa	0.7	0	Вредоносный набор: псевдополиморфные образцы, собранные средством Polygraph; полиморфные версии червя CodeRed2 [16]; код, сгенерированный CLET [41],TAPiON [17]; Нормальный трафик: HTTP URI
Polygraph	0.2	0	Вредоносный набор: Apache-Knacker [29], эксплойт ATPhttpd [45], BIND-TSIG [46]; 125301 потоков 10-ти дневной HTTP трассы
Stride	0.0027	0	Вр. набор: NOP-следы, сгенерированные Metasploit Framework v2.2 [10]; Нормальный трафик: HTTP URI;
Racewalk	0.0058	0	Вр. набор: NOP-следы, сгенерированные Metasploit Framework v2.2 [10]; Н.трафик: HTTP URI, ELF-файлы, ASCII, мультимедия, псевдо-рандомные данные.
Styx	0	0	Вр. набор: эксплойты, сгенерированные Metasploit Framework v2.2 [10]; Н.трафик: трафик, собранный с рабочей сети: Windows- и linux- хосты

Таблица 3.1: Значения ошибок первого и второго рода и описание тестового набора данных некоторых методов

Метод	FP, %	FN, %	Тестовые данные
Structural	0.5	0	Вр. набор: эксплойты, сгенерированные средством ADMmutate [61]; Н.трафик: HTTP (45%), SMTP (35%), а так трафик приложений: SSH, IMAP, DNS, FTP.
SigFree	0	0	Вр. набор: незашифрованные атаки, сгенерированные Metasploit framework, червь Slammer, CodeRed Н.трафик: HTTP ответы (шифрованные данные, аудио, jpeg, png, gif и flash).
STILL	0	0	Вр. набор: эксплойты, сгенерированные Metasploit, CLET, ADMmutate
SAMD	0	0	Вр. набор: обфусцированные варианты червя B[e]agle [3]; Н.трафик: 2,000 Windows программ
Emulation	0.004	0	Вр. набор: эксплойты, сгенерированные Clet, ADMmutate, TAPiON и Metasploit framework; Н.трафик: рандомные бинарные данные
HDD	0.0126	0	Вр. набор: эксплойты, сгенерированные Metasploit, CLET и ADMmutate Н.трафик: UDP, FTP, HTTP, SSL; бинарные файлы Windows

Таблица 3.2: Значения ошибок первого и второго рода и описание тестового набора данных некоторых методов

Метод	FP, %	FN, %	Тестовые данные
NSC	0	0	Вр. набор: эксплойты, сгенерированные модулями Avoid UTF8, Encoder и Alpha2 средства Metasploit Framework; Н.трафик: бинарные данные, ASCII данные.
IGPSA	0	0	Вр. набор: эксплойты, сгенерированные Clet, ADMmutate, TAPiON и Metasploit framework, Jempiscodes [90]; Н.трафик: трассы HTTP и HTTPS - порты: 80, 443, 135, 139 и 445
PolyUnpack	0	0	Вр. набор: 3,467 образцов из репозитория вредоносного кода OARC.
APE	0	0	Вр. набор: IIS 307, 1887 эксплойтов BID Н.трафик: HTTP и DNS запросы.

Таблица 3.3: Значения ошибок первого и второго рода и описание тестового набора данных некоторых методов

Method	Complexity	Remarks
Buttercup	$O(N)$	N - длина проверяемого объекта S
Hamsa	$O(T \times N)$	N - длина S, T - число токенов
Polygraph	$O(N)$	без кластеризации: N - длина S
	$O(N + C^2)$	с кластеризацией: C - количесво кластеров
	$O(M^2 \times L)$	обучение: M - длина вред. выборки,
		L - длина легитимной обучающей выборки
Stride	$O(N \times l^2)$	N - длина S, l - длина NOP-следа
Racewalk	$O(N \times l)$	N - длина S, l - длина NOP-следа
Styx	$O(N)$	N - длина проверяемого объекта S
Structural	$O(N)$	N - длина проверяемого объекта S
SigFree	$O(N)$	N - длина проверяемого объекта S
STILL	$O(N)$	N - длина проверяемого объекта S
SAMD	$O(N)$	N - длина проверяемого объекта S
Emulation	$O(N^2)$	N - длина проверяемого объекта S
HDD	$O(N + K^2 \times T^2)$	N - длина S, K - количество подозр. цепочек,T - длина максимальной подозрительной цепочки
NSC	$O(N^2)$	N - длина проверяемого объекта S
Sandbox	$O(N)$	N - длина проверяемого объекта S
IGPSA	$O(N^2)$	в неоптимизированном варианте
	$O(CN)$	в оптимизированном
PolyUnpack	$O(N)$	N - длина проверяемого объекта S
APE	$O(N \times 2^l)$	N - длина проверяемого объекта S
		l - длина NOP-следа

Таблица 3.4: Алгоритмическая сложность методов

	Buttercup	Hamsa	Polygraph	Stride	Racewalk	Styx	Structural analysis	SigFree	STILL	SAMD	Emulation	HDD	NSC	Sandbox	IGPSA	PolyUnpack	APE
K_{NOP-PL}		√	√	√	√			√	√								√
K_{NOP-B}		√	√	√	√			√	√								√
K_{NOP-MB}		√	√	√	√			√	√								√
$K_{NOP-4AL}$		√	√	√	√			√	√								√
K_{NOP-T}		√	√	√	√			√	√								√
K_{NOP-TO}		√	√	√	√			√	√								
$K_{NOP-SAR}$						√		√	√		√	√				√	
K_{GET-PC}						√			√		√	√		√	√	√	
K_{PL}	√	√	√				√	√	√	√				√			
K_{DATA}	√	√	√				√	√	√	√				√			
K_{ALT_OP}	√	√	√				√	√	√	√				√			
K_R	√							√		√				√			
K_{ALT_INS}	√						√	√	√					√			
K_{INJ}	√		√				√	√	√	√				√			
K_{SU}						√			√		√	√		√	√	√	
K_{SC}						√			√		√	√		√	√	√	
K_{RET}	√						√	√	√	√							
K_{RET_+}		√					√	√	√	√							
K_{MET}							√		√		√	√		√	√	√	
K_{NSC}													√			√	

Таблица 3.5: Покрытие классов вредоносного исполнимого кода рассматриваемыми методами

пускная способность рассматриваемых методов, делает возможной их применимость к высокоскоростным каналам передачи данных, минимизируя задержку анализируемых данных. Тем не менее, даже невысокая величина процента ложных срабатываний (ошибки второго рода) приводит к большому абсолютному значению количества ошибок. При больших объемах анализируемых данных это значение может привести к отказу в обслуживании легитимных пользователей.

2. *Методы, обладающие невысоким процентом ложных срабатываний, характеризуются высокой вычислительной сложностью.* Применение таких методов к высокоскоростным каналам в режиме реального времени невозможно. Тем не менее, методы могут использоваться в качестве дополнительной проверки методов, работающих быстро, но обладающих высоким процентом ложных срабатываний. Кроме того, данные методы можно использовать в режиме «офф-лайн», где ограничения на время выполнения не настолько строгие.
3. *Ни один из существующих методов обнаружения вредоносных исполнимых инструкций не обеспечивает полного покрытия классов вредоносных инструкций.* Другими словами, каждый из рассмотренных методов в состоянии обнаруживать один или несколько классов вредоносных исполнимых инструкций. В случае применения такого выбора к реальному каналу передачи данных, целые семейства вредоносного кода не будут обнаружены, и. несмотря на то, что риск передачи таких инструкций по каналу будет снижен, вероятность его появления не будет нулевой. Тем не менее, стоит заметить, что *комбинация* части рассмотренных алгоритмов может обеспечить полное покрытие классов вредоносных инструкций, однако это связано со значительным количеством вычислительных затрат.

Таким образом, актуальна задача разработки классификатора, обладающего следующими свойствами:

1. Классификатор обеспечивает полное покрытие выделенных классов вредоносного исполнимого кода;
2. Классификатор в среднем случае обладает меньшей вычислительной сложностью, чем линейная комбинация входящих в него методов;
3. Классификатор минимизирует долю ложных срабатываний входящих в него методов.

В главах 2 и 3 мы показали, что задача обнаружения шеллкодов в сетевом трафике удовлетворяет модели, введенной в главе 1: в качестве объектов $\mathfrak{s}_j$ выступают дизассемблированные байтовые строки, был введен набор признаков объектов, была предложена классификация шеллкодов и описан набор классификаторов. Таким образом, задача разработки классификатора, обнаруживающего шеллкоды в сетевом трафике, может быть решена подходом, предложенным в главе 1.

Глава 4. Инструментальная среда обнаружения шеллкодов DEMORPHEUS

В данной главе представлено описание прототипа средства обнаружения шеллкодов, названного Demorpheus. Прототип реализует решение задачи обнаружения шеллкодов, предложенное в главе 1. Прототип реализован на двух языках программирования - C и C++. На момент написания текста диссертационной работы прототип содержал 11873 строки кода. Исходные коды прототипа доступны в открытом доступе по адресу https://gitorious.org/demorpheus.

4.1. Архитектура

Средство обнаружения шеллкодов Demorpheus поддерживает два основных режима работы: работа в режиме анализа сетевого трафика и работа в режиме анализа переданных системе на вход файлов. В случае, если средство запущено в первом режиме работы, Demorpheus считывает входные данные с указанного сетевого интерфейса непосредственно. Во втором случае, происходит обработка и анализ входного файла. Результатом работы средства является выделение из переданного объема входных данных объектов, носящих вредоносных характер, и их классификация.

На рисунке 4.1 приведена схема архитектуры Demorpheus. Прямоугольниками обозначены компоненты средства, стрелками отображен поток данных.

Множество входных форматов средства: поток, считываемый с сетевого интерфейса, или файл любого формата. Файлы рассматриваются средством как байтовый поток данных.

Модуль *"Сбор сетевой сессии"* отвечает за восстановление потока данных из пакетов, считываемых с сетевого интерфейса. Модуль запускает-

Рис. 4.1: Инструментальная среда Demorpheus.

ся только в том случае, если средство запущено в режиме анализа сетевого трафика.

Модуль *"Дизассемблирование входного потока"* преобразует входной байтовый поток в набор инструкций целевого процессора. Средство поддерживает три набора команд процессоров: набор команд платформы x86 и набор команд платформы ARM с возможностью переключения в режим Thumb.

Модуль *"Восстановление служебных структур"* преобразует набор инструкций ассемблера в некоторые служебные структуры, требуемые для работы средства: граф потока управления, граф потока инструкций, набор потоков, дизассемблированных с каждого смещения.

Модуль *"Библиотека шеллкодов"* содержит набор классификаторов. Часть классификаторов в библиотеке содержит по одному алгоритму, определяющему один из признаков шеллкодов, перечисленных в главе 2. Часть

классификаторов представляет из себя декомпозированных алгоритмов обнаружения шеллкодов. Часть классификаторов в библиотеке являются оригинальными методами обнаружения шеллкодов, описанными в главе 3. Каждый классификатор имеет универсальный интерфейс для запуска гибридным классификатором.

Модуль *"Гибридный классификатор"* осуществляет выполнение оптимальной топологии классификаторов, предоставляемых библиотекой шеллкодов. Кроме того, модуль поддерживает режим работы, в котором осуществляется единовременное построение оптимальной топологии. Построение топологии осуществляется по алгоритму, описанному в главе 1.

На рисунке 4.2 представлен пример работы инструментальной среды Demorpheus в случае анализа сетевого трафика. При обнаружении образца шеллкода, средство показывает уведомление об этом, указывая классы шеллкодов, к которым относится обнаруживаемый образец.

Рис. 4.2: Пример работы среды Demorpheus.

4.2. Компоненты системы

В данном разделе подробно рассмотрено устройство компонент средства Demorpheus.

4.2.1. Дизассемблирование входного потока

Шеллкод может содержаться во входной байтовой строке, начиная с любого смещения. Таким образом, для анализа переданного на вход образца данных недостаточно одного прохода средства дизассемблирования. Модуль дизассемблирования входного потока является одним из наиболее критичных с точки зрения пропускной способности средства: из-за необходимости в общем случае проводить анализ $n-1$ потока для данных, содержащего n байт, вычислительная сложность модуля дизассемблирования существенно влияет на производительность всего средства.

На сегодняшний день существует ряд средств дизассемблирования, таких как IDA Pro [42], OllyDebug [12], Boomerang [4], REC [13], Netwide Disassemler [11], diStorm [5]. Данные средства используют различные техники дизассемблирования: линейное дизассемблирование, рекурсивное дизассемблирование и другие [88]. Как правило, алгоритмическая сложность указанных средств дизассемблирования оценивается значением $O(Cn)$, где n - размер анализируемой байтовой строки, а C - некоторая константа, которая оценивает сложность поиска корректной инструкции процессора, соответствующей анализируемому коду операции, во внутренней структуре данных конкретного средства дизассемблирования. Как правило, сложность поиска инструкции линейна по количеству инструкций процессора, а в виду ограниченности их числа, оценивается константным значением.

Несмотря на линейную сложность существующих средств дизассемблирования, возможно улучшить значения константы до показателя $O(ln(m))$, где m - число команд процессора. В целях оптимизации вычислительной сложности дизассемблирования, в средстве Demorpheus модуль дизассемблирования реализован на основе префиксного дерева, описанного в работе [99]. Префиксное дерево генерируется однократно. Дерево представляет из себя иерархическую структуру, в которой корень является пу-

стым словом. Вершины дерева соответствуют некоторым байтовым значениям, входящих в состав операционного кода инструкции. Например, команда `ADC$0x??, %cl`, код операции которой состоит из двух байт `x80xDL`, представляется двумя вершинами префиксного дерева. Путь от корня до листа дерева соответствует операционному коду некоторой команды целевого процессора. Пример префиксного дерева для двух команд процессора IA-32 приведен на рисунке 4.3. Операционный код `x37` соответствует инструкции `AAA`, операционный код `x80xDL` соответствует инструкции `ADC$0x??, %cl`.В данной работе в префиксном дереве были использованы инструкции процессора IA-32 [35]. Для дизассемблирования потока данных в команды ARM используются библиотеки armstorm [2] и libdisarm [9].

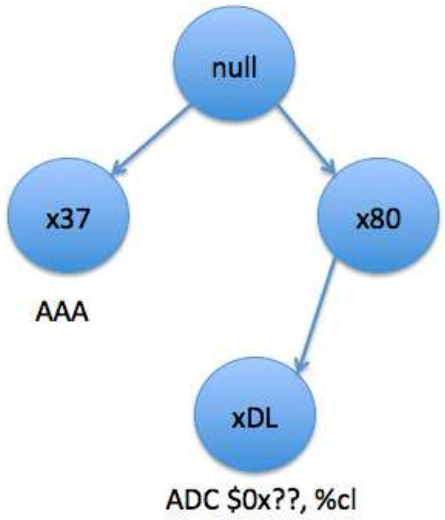


Рис. 4.3: Пример части префиксного дерева дизассемблера.

4.2.2. Восстановление служебных структур

Данный модуль строит несколько служебных структур из дизассемблированного потока входных данных. Эти структуры используются классификаторами библиотеки шеллкодов для анализа входных объектов и заключения об их вредоносности. К таким структурам относится вектор це-

почек дизассемблированных инструкций анализируемого потока данных, граф потока управления и граф потока инструкций.

Вектор цепочек дизассемблированных инструкций (далее - ВЦДИ) представляет из себя наборы инструкций, полученные путем дизассемблирования входного потока с разных смещений. Такая структура необходима в виду того, что тело шеллкода может начинаться с любого смещения входного потока. Каждая инструкция в векторе представима структурой, содержащей следующие значения:

- Размер инструкции. Размер инструкций процессора IA-32 варьируется от 1 байта до 15 байт [35], включая размер операндов. Эта информация может требоваться для корректного дизассемблирования последующих инструкций или иной обработки входного потока данных.
- Тип инструкции. Инструкция может относиться к одному из 106 типов, перечисленных в [35]. Примером таких типов могут служить инструкции безусловной передачи управления (`INSTRUCTION_TYPE_JMP`), инструкции условной передачи управления (`INSTRUCTION_TYPE_JMPC`) и другие.
- Смещение, с которого начинается инструкция во входном потоке данных. Этот параметр характеризует относительный адрес начала инструкции во входном потоке.
- Информация об операндах инструкции в случае их наличия. К такой информации относится тип операнда, регистр (если операнд имеет тип регистрового значения) и абсолютное значение операнда (если операнд имеет соответствующий тип).

Если очередная инструкция в цепочке попадает на уже проанализированное смещение, в вектор помещается ссылка на элемент, содержащий инструкцию с заданным смещением. В качестве примера рассмотрим ВЦДИ, полученный из байтовой строки `e2 08 eb 06 eb 04 d4 5e 1c fd`. В

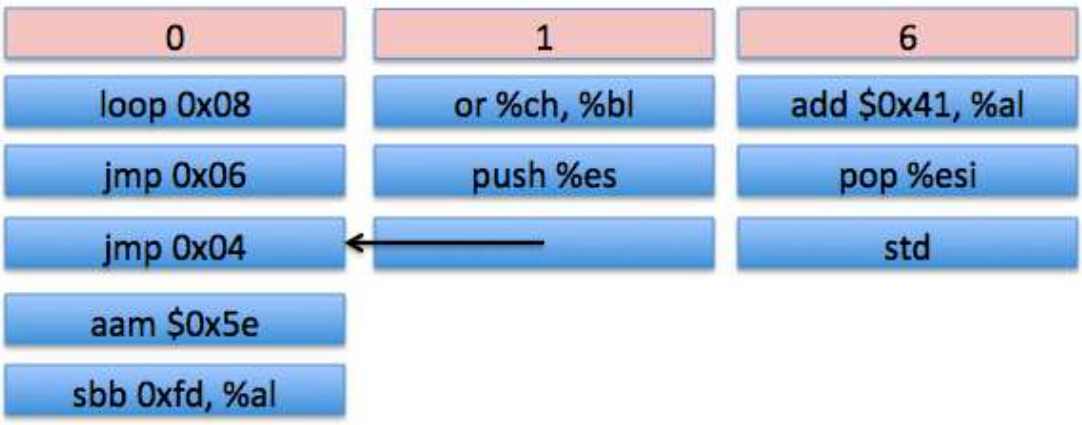


Рис. 4.4: Пример вектора дизассемблированных цепочек.

результате дизассемблирования со смещения 0, будет получен набор инструкций, соответствующих левому столбцу рисунка 4.4. В результате дизассемблирования со смещения 1, будет получена цепочка инструкций, соответствующая второму столбцу. При этом, полученная цепочка, начиная с третьего элемента, совпадает с подцепочкой цепочки инструкций, полученной путем дизассемблирования со смещения 0. Таким образом в третий элемент цепочки, начинающейся со смещения 1, будет помещена ссылка на третий элемент цепочки, начинающейся со смещения 0. Цепочки, полученные путем дизассемблирования со смещений 2-4, совпадают с подцепочками цепочек, разобранных ранее.

Восстановление IFG. IFG (instruction flow graph) представляет из себя направленный граф, вершины которого представляют из себя инструкции, а дуги соответствуют возможной передаче потока управления компилируемой программы. Граф потока инструкций является частным случаем графа потока управления CFG (control flow graph). Граф потока управления представляет из себя структуру данных, абстрагирующую поведение потока управления компилируемой программы [7]. Граф потока управления является направленным, вершины которого представляют из себя базовые блоки программы, а дуги соответствуют возможной передаче управле-

ния от одного базового блока другому. Базовый блок - последовательность инструкций, имеющая один вход, один выход и не содержащая инструкций передачи управления. Существует большое количество решений задачи восстановления графа потока управления и, соответственно, графа потока инструкций из байтовой строки. В качестве примера можно привести работы [66], [98], [26], [63], [51] и [65]. Тем не менее, в контексте данной работы, задача восстановления графа потока управления (графа потока инструкций) имеет несколько особенностей:

1. Необходимость восстановления графов, начиная с каждого смещения входного потока.
2. В целях оптимизации вычислительной сложности всей системы оправдано использование результатов предыдущего шага дизассемблирования и восстановления ВЦДИ.

Исходя из заданных ограничений, в работе был предложен и реализован алгоритм восстановления графа потока управления из байтовой строки.

Подробное описание алгоритма и его анализ приведены в Приложении А.

Восстановление CFG.

Для восстановления графа потока управления используется алгоритм восстановления графа потока инструкций с незначительными модификациями. При восстановлении CFG вызов функции `makeConnection` осуществляется при анализе инструкции передачи управления. Все остальные инструкции сохраняются в структуру базового блока, которыми и оперирует алгоритм.

4.2.3. Библиотека шеллкодов

Библиотека шеллкодов представляет из себя набор классификаторов шеллкодов с универсальным интерфейсом.

1. Классификаторы, содержащие по одному алгоритму, проверяющему наличие во входном объекте заданные признаков шеллкодов, подробно описанных в Главе 2.
2. Декомпозированные методы обнаружения шеллкодов. Существующие методы обнаружения шеллкодов используют для своего анализа набор из одного или нескольких алгоритмов, обнаруживающих признаки шеллкодов. В библиотеке такие алгоритмы или их некоторая комбинация, в случае, если алгоритмы зависимы, реализованы в виде классификаторов. Ряд существующих методов обнаружения шеллкодов используют часть общих алгоритмов, что позволяет существенно снизить суммарную вычислительную сложность системы путем однократного запуска общих алгоритмов на входном потоке.
3. Методы обнаружения шеллкодов. Часть существующих методов обнаружения не декомпозируются на независимые алгоритмы. В этом случае методы содержатся в библиотеке в исходном варианте.

Классификаторы шеллкодов производят анализ одной или нескольких служебных структур, восстановленных из входного объекта. Факт необходимости анализа сразу нескольких структур одним классификатором объясняется тем, что состав алгоритмов, входящих в него, может быть гетерогенен. Под гетерогенностью здесь будем понимать возможность осуществления как статического, так и динамического анализа, а так же возможностью анализа различных служебных структур. После анализа классификатор меняет соответствующие биты информационного вектора объекта.

Каждый классификатор в библиотеки шеллкодов характеризуется тремя параметрами - долей ошибок первого рода, характеризующей точность обнаружения классификатора; долей ошибок второго рода (ложными срабатываниями) и вычислительной сложностью. Средство Demorpheus позволяет вычислять эти параметры автоматически, запуская классификаторы на некотором тестовом наборе-бенчмарке. Для вероятности оши-

бок первого и второго рода вычисляются абсолютные значения в диапазоне $[0, 1]$.

Сложность классификатора определяется относительным значением в диапазоне $[0, 1]$ по следующей формуле:

$$C(\mu) = \frac{tp(\mu) - min\_tp}{max\_tp - min\_tp}, \tag{4.1}$$

где μ - заданный классификатор, $tp(\mu)$ - пропускная способность заданного классификатора, max_tp - максимальная пропускная способность среди всех доступных классификаторов, min_tp - минимальная пропускная способность среди всех доступных классификаторов. Другими словами, сложность классификатора представляет из себя некоторое нормализованное значение в диапазоне минимальной и максимальной сложности всех имеющихся классификаторов.

4.2.4. Гибридный классификатор

Модуль *"Гибридный классификатор"* средства Demorpheus подразумевает два режима работы. В первом режиме однократно генерируется статического дерево классификаторов с оптимальной топологией. Модуль гибридного классификатора должен быть запущен в данном режиме каждый раз при добавлении новых классификаторов в библиотеку шеллкодов.

В данном режиме работы модуль анализирует все доступные классификаторы библиотеки шеллкодов, извлекая присущие им значения вероятности ошибок второго рода, сложности и обнаруживаемых классификаторами классов шеллкодов. Далее модуль выстраивает дерево классификатора по алгоритму, подробно рассмотренному в Главе 1. Дерево представляется в виде статического объекта, загружаемого средством при каждом запуске.

Во втором режиме работы модуль *"Гибридный классификатор"* осуществляет выполнение оптимальной топологии классификаторов для каждого поступившего на анализ входного объекта. Модуль контролирует пе-

редачу управления между различными классификаторами, анализируя результаты их работы на входном потоке. В соответствии с описанием подхода, лежащего в основе данного модуля (см. Главу 1), принимается решение о запуске или незапуске конкретных классификаторов, входящих в структуру, на заданном входном потоке. После завершения работы классификаторов модуль анализирует информационный вектор входного объекта и принимает решение о его вредоносности или легитимности.

4.3. Испытания прототипа

Испытания прототипа средства обнаружения шеллкодов проводились на виртуальной машине, на которой была запущена операционная система Ubuntu версии 10.1. Тестовая среда обладала следующими характеристиками: Intel Core 2 Duo CPU, 2.53 HGz, 4 GB RAM.

4.3.1. Тестовые наборы данных

Апробация прототипа средства обнаружения шеллкодов проводилась на четырех различных наборах данных:

1. *Набор шеллкодов.* Для апробации прототипа было использовано 1536 образцов шеллкодов, сгенерированных средством автоматического генерации вредоносного исполнимого кода Metasploit [10], а так же 42 образца шеллкодов, доступных в публичном репозитории эксплойтов [6]. При генерации образцов средством Metasploit были использованы все доступные модули обфускации, входящие в средство. В процессе проведения экспериментов, 1578 образцов шеллкодов были подмешаны в нормальный трафик. Средство тестировалось в режиме анализа трафика. Эксперименты, проводимые на указанном наборе данных, являются показательными для оценки точности работы тестируемого прототипа.

2. *Набор легитимных бинарных файлов.* Один из наиболее показательных экспериментов, оценивающих качество работы прототипа с точки зрения ложных срабатываний - набор легитимных бинарных файлов. С одной стороны, образцы тестового набора представляют из себя корректно исполнимые программы, и, соответственно, обладают признаками исполнимых программ, которые в числе прочих проверяются классификаторами шеллкодов. С другой стороны, данные программы не являются вредоносными и, соответственно, не обладают специфичными признаками, присущими шеллкодам (Глава 2). В данном тестовом наборе было использовано 200 исполнимых PE Windows файлов и 200 исполнимых ELF-файлов unix-подобных операционных систем, содержащихся в директории `/usr/bin`. При тестировании на данном тестовом наборе, средство использовалось в режиме анализа файлов.
3. *Набор случайных данных.* Для оценки значения вероятности ошибок второго рода средство так же тестировалось на случайно сгенерированных данных. Для проведения тестов был использован тестовый набор, содержащих 300 Мб данных.
4. *Набор мультимедийных данных.* Такие данные представляют из себя набор мультимедийных файлов, таких как рисунки и фотографии (`.jpg`, `.png`), а так же музыка и видео. Такой набор данных характеризует качество алгоритмов с точки зрения доли ложных срабатываний. В процессе проведения экспериментов было замечено, что многие из существующих алгоритмов (см. Главу 3) имеют значительную долю ложных срабатываний (например, до 47 % в случае Polygraph) на данном тестовом наборе.

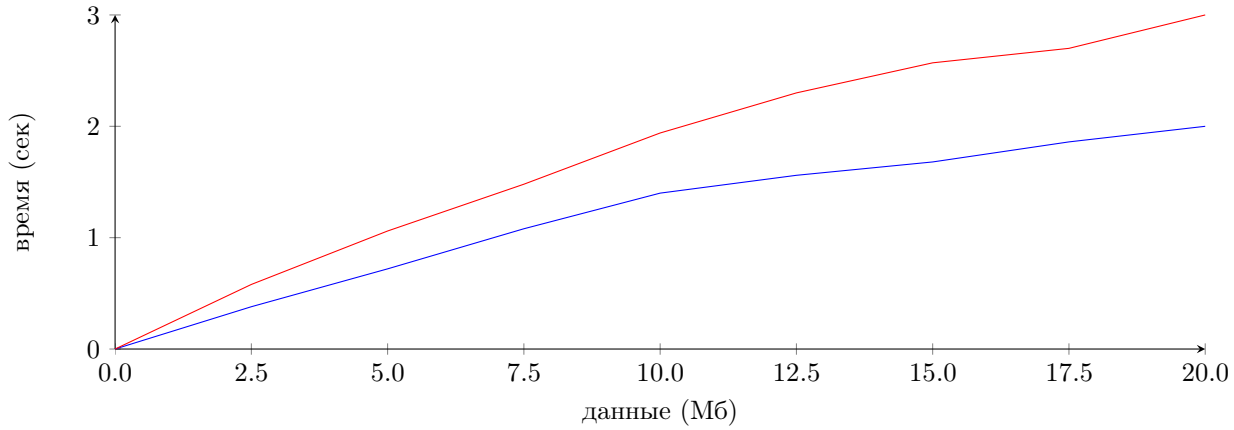
4.3.2. Результаты экспериментов

При проведении экспериментов осуществлялось сравнение гибридного классификатора с линейной структурой классификаторов, так же описанной в разделе 1.2 главы 1. Как было показано, такая структура характеризуется минимальным значением вероятности ошибок. В данном случае под ошибками будем понимать ложные срабатывания алгоритма (ошибки второго рода). Две различные структуры - гибридная и линейная - сравнивались по пропускной способности, вероятности ошибок первого и второго рода. Несмотря на то, что при построении гибридной структуры ошибки первого рода не учитывались, данный парамент важен для понимания, как выбор структуры влияет на точность обнаружения шеллкодов.

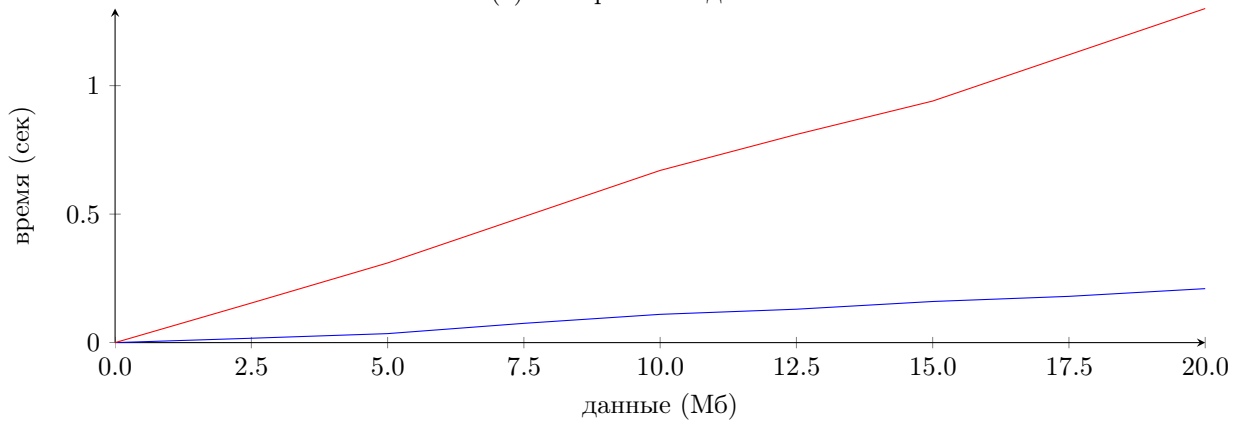
Результаты экспериментов для сравнения ошибок первого и второго рода, а так же пропускной способности на тестовой виртуальной машине, приведены в таблицах 4.1, 4.2 и 4.3, соответственно. Для тех наборов данных, для которых вычисление ошибки первого или второго рода не имеет смысла, это значение принимает вид n/a. В качестве примера можно рассмотреть ошибку первого рода и набор легитимных данных. Ошибка первого рода - это принятие вредоносного потока за легитимный. Очевидно, что при тестировании на легитимных данных вычисление такой ошибки невозможно.

Как видно из приведенных таблиц, очевиден компромисс между сложностью алгоритма, выраженной в данном случае пропускной способностью, и долей ложных срабатываний. Значение доли ложных срабатываний для гибридной структуры классификатора несколько выше, чем для линейной топологии, однако очевидно значительное преимущество гибридной структуры с точки зрения пропускной способности.

Результаты экспериментов по оценке доли ошибок первого рода более интересны. С одной стороны, предлагаемый подход помещает наиболее



(a) Набор шеллкодов

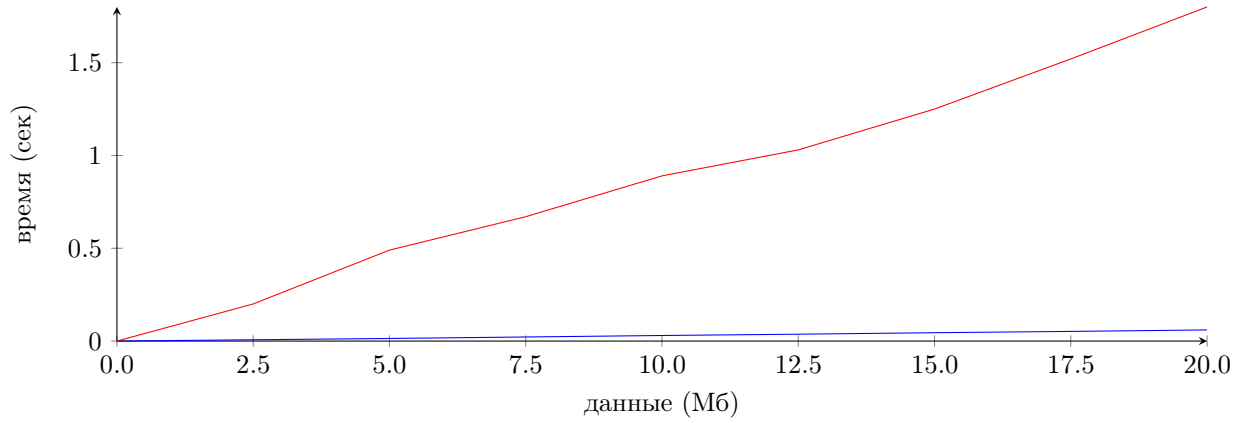


(b) Набор легитимных программ

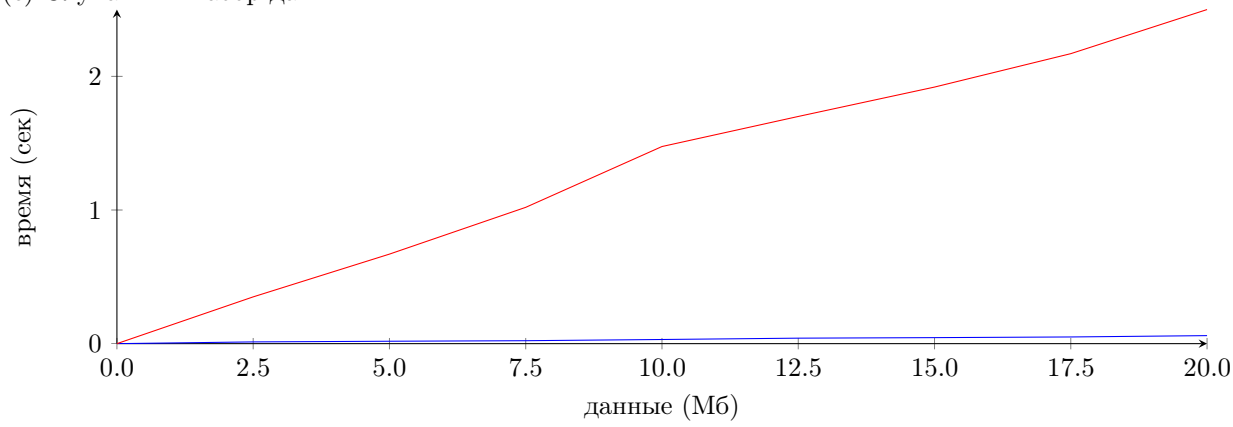
Рис. 4.5: Сравнение времени работы для линейной и гибридной топологии детекторов на вредоносном и легитимном наборе данных. Красная линия соответсвует линейной топологии, голубая - гибридной.

простые с точки зрения вычислительной сложности классификаторы на верхние уровни топологии. Теоретически, такие классификаторы являются менее устойчивыми к техникам уклонения от обнаружения. Из-за этого до проведение экспериментов предполагалось более высокое значение доли ошибок первого рода. Тем не менее, результаты экспериментов опровергают интуитивное предположение. Этот факт объясняется следующим:

- "Простые"классификаторы, обладающие невысокой вычислительной сложностью, в реализованном прототипе, как правило, осуществляют проверку наличия во входном потоке универсальных признаков, присущих большому количеству классов шеллкодов. Типичный пример -



(c) Случайный набор данных



(d) Мультимедийный набор данных

Рис. 4.6: Сравнение времени работы для линейной и гибридной топологии детекторов на случайном и мультимедийном наборе данных. Красная линия соответсвует линейной топологии, голубая - гибридной.

проверка, содержится ли в потоке некоторый минимальный набор исполнимых инструкций или нет. Такой минимальный набор инструкций необходим, например, для организации декрипторной части шеллкода в случае его полиморфной природы. Очевидно, что подобные классификаторы характеризуются значительным числом ложных срабатываний, в виду того, что проверяемые ими характеристики применимы и к большому количеству легитимных образцов кода. Тем не менее, такие классификаторы устойчивы с точки зрения ошибок первого рода.

- Так как каждый уровень гибридного классификатора предоставля-

Набор данных	Линейная структура	Гибридная структура
шеллкоды	0.2	0.204
легитимные программы	n/a	n/a
случайные данные	n/a	n/a
мультимедия	n/a	n/a

Таблица 4.1: Результаты значений ошибок первого рода (FN) для линейной и гибридной структур классификаторов.

Набор данных	Линейная структура	Гибридная структура
шеллкоды	n/a	n/a
легитимные программы	0.0064	0.019
случайные данные	0	0
мультимедия	0.005	0.04

Таблица 4.2: Результаты значений ошибок второго рода (FP) для линейной и гибридной структур классификаторов.

ет полное покрытие классов шеллкодов, которые способны обнаруживать классификаторы библиотеки шеллкодов. Как правило, на каждом уровне гибридной топологии классификатора, оказывается сразу несколько классификаторов. В этом случае входной поток будет проанализирован несколькими цепочками классификаторов, результат выполнения которых не зависит друг от друга.

Сравнение результатов времени выполнения линейной и гибридной структур классификаторов для разных тестовых наборов приведены на рисунках 4.5 и 4.6. Несмотря на то, что абсолютное значение пропускной способности не велико, оно может быть легко увеличено за счет увеличения мощности тестового оборудования. Так же возможен параллельный запуск

Набор данных	Линейная структура	Гибридная структура
шеллкоды	6.9	11
легитимные программы	15	92.6
случайные данные	11	320
мультимедия	8	325

Таблица 4.3: Результаты значений пропускной способности на тестовой машине. Значение оценивается в Мб/сек.

средства на нескольких вычислительных ресурсах для получения требуемой пропускной способности - например, на 3-10 для пропускной способности в 1Гб/сек на оборудовании, аналогичном тестовому.

Сравнение относительного времени работы линейной структуры и гибридной структуры показывает, что гибридная структура в разы эффективнее. На вредоносном наборе тестовых данных скорость работы гибридной структуры в несколько выше скорости работы линейной. Это объясняется тем фактом, что даже на вредоносном наборе данных часть классификаторов гибридной структуры запущена не будет. К примеру, если входной поток является шеллкодом некоторого класса K_1, он будет рассмотрен как легитимный классификаторами, не обнаруживающими данный класс. Кроме того, остается доля ошибок первого рода. Тем не менее, ошибки первого рода компенсируются запуском других классификаторов, а суммарное время работы классификатора снижается. На наборе легитимных программ гибридная структура демонстрирует 8-ми кратное преимущество перед линейной структурой классификаторов. Стоит отметить, что это значение ниже, чем преимущество гибридной структуры на наборе случайных данных, достигающее 34-х кратного значения. Это подтверждает утверждение о том, что легитимные программы содержат признаки, присущие и шеллкодам. Таким образом, входной поток, состоящий из легитимного набора программ, не бу-

дет отсечен на самых верхних уровнях классификатора. Наиболее большую разницу с линейной структурой в скорости работы гибридная структура демонстрирует на мультимедийном наборе данных. Эта разница достигает значения 40-ка раз.

Глава 5. Заключение

В работе ставилась задача исследования и разработки метода обнаружения вредоносного исполнимого кода, эксплуатирующего ошибки работы с памятью, в высокоскоростных каналах передачи данных. Кроме того, ставилась задача разработать и апробировать прототип инструментальной среды обнаружения вредоносного исполнимого кода.

В диссертационной работе получены следующие результаты, характеризующиеся научной новизной.

1. Проведен анализ существующих шеллкодов и разработана их классификация, позволяющая полностью покрыть все известные образцы шеллкодов.
2. Предложена модель распознавания объектов, позволяющая обнаруживать в объектах набор специфичных признаков и распознавать объекты в соответствии с предложенной классификацией. В рамках представленной модели разработан алгоритм распознавания шеллкодов, позволяющий покрыть все известные классы шеллкодов; снизить вычислительную сложность обнаружения шеллкодов; минимизировать количество ложных срабатываний.
3. Разработанные алгоритмы классификации были реализованы и апробированы в рамках экспериментальной системы Demorpheus на четырех наборах данных: на наборе эксплойтов, на легитимных программах, на случайных и мультимедийных данных. Система продемонстрировала практически нулевое число ложных срабатываний, высокое значение пропускной способности по сравнению с линейной комбинацией существующих аналогов.

Инструментальная среда Demorpheus, являясь расширяемой, позволяет добавлять новые алгоритмы обнаружения шеллкодов, а так же расширять множество обнаруживаемых классов шеллкодов. Инструментальная

среда Demorpheus позволяет обнаруживать шеллкоды как в сетевом трафике, так и в любых файлах.

В будущем планируется учитывать вероятность появления исполнимых файлов в потоке данных для динамического перестроения алгоритмов, входящих в состав инструментальной среды Demorpheus. Так же планируется расширить множество обнаруживаемых классов шеллкодов, в частности, ROP-шеллкодов.

Литература

1. Arbor networks: Ddos protection, prevention and mitigation // http://atlas.arbor.net/.
2. The armstorm project // https://code.google.com/p/armstorm/.
3. The bagle botnet // https://www.securelist.com/en/analysis/162656090/the_bagle_botnet.
4. Boomerang project. http://boomerang.sourceforge.net/.
5. distorm project. http://www.ragestorm.net/distorm/.
6. Exploit database // http://www.exploit-db.com/.
7. *A GNU manual*, chapter Control Flow Graph. Free Software Foundation, Inc.
8. Inside the ddos botnets - blackenergy and darkness, http://extreme-security.blogspot.ru/2012/08/inside-ddos-botnets-blackenergy-and_27.html.
9. libdisarm documentation // http://iriver-t10.sourceforge.net/libdisarm-api.html.
10. Metasploit: Penetration testing software. http://www.metasploit.com/.
11. The netwide assembler project. http://www.nasm.us/.
12. Ollydebug . http://www.ollydbg.de/.
13. Rec studio 4 - reverse engineering compiler. http://www.backerstreet.com/rec/rec.htm.
14. Snort project. snort.org.
15. A story of a spam botnet cutwail trojan - via fake paypal's spam link w/redirector (92.38.227.2) backboned by bhek2 (80.78.247.227), http://malwaremustdie.blogspot.ru/2013/05/a-story-of-spambot-trojan-via-fake.html.

16. Symantec security response: Codered worm. http://www.sarc.com/avcenter/venc/data/codered.worm.html.
17. Tapion project. *http://pb.specialised.info/all/tapion/*.
18. P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis. Stride: Polymorphic sled detection through instruction sequence analysis. In *In 20th IFIP International Information Security Conference*, 2005.
19. Esraa Alomari, Selvakumar Manickam, B. B. Gupta, Shankar Karuppayah, and Rafeef Alfaris. Botnet-based distributed denial of service (ddos) attacks on web servers: Classification and art. *CoRR*, abs/1208.0403, 2012.
20. S. Andersson, A. Clark, and G. Mohay. Detecting network-based obfuscated code injection attacks using sandboxing. In *AusCERT Asia Pacific Information Security Conference*, Gold Coast, Australia, 2005.
21. M. Marquez Andrade and N. Vlajic. Dirt jumper: A new and fast evolving botnet-for-ddos. volume 3, pages 330–336.
22. Anonymous. Once upon a free(). *Phrack Magazine*, 57(9), 2001.
23. B. AsSadhan, J.M.F. Moura, David Lapsley, C. Jones, and W.T. Strayer. Detecting botnets using command and control traffic. In *Network Computing and Applications, 2009. NCA 2009. Eighth IEEE International Symposium on*, pages 156–162, 2009.
24. L. Babai and E. Luks. Canonical labeling of graphs. *15th ACM Symposium on Theory of Computing*, 1983.
25. P. Barford and V. Yegneswaran. An inside look at botnets. *Advances in Information Security*, 27:171–191, 2007.
26. N. Bermudo, A. Krall, and N. Horspool. Control flow graph reconstruction for assembly language programs with delayed instructions. In *Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on*, pages 107–116, 2005.
27. Blexim. Basic integer overflows. *Phrack Magazine*, 60(10), 2002.

28. Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. pages 78–92. Springer-Verlag, 2002.
29. C. CAN-2003-0245. Apache apr-psprintf memory cor- ruption vulnerability. *http://www.securityfocus. com/bid/7723/discussion/.*
30. Rich Caruana and Alexandru Niculescu-mizil. An empirical comparison of supervised learning algorithms. In *In Proc. 23 rd Intl. Conf. Machine learning (ICML'06*, pages 161–168, 2006.
31. E. Chien. Technical report: W32.stuxnet dossier, symantec. 2010.
32. Ramkumar Chinchani and Eric Van Den Berg. A fast static analysis approach to detect exploit code inside network flows. In *In Proceedings of the 8 th International Symposium on Recent Advances in Intrusion Detection (RAID*, pages 284–304, 2005.
33. Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46, Washington, DC, USA, 2005. IEEE Computer Society.
34. Thomas Cormen, Charles Leiserson, Raonald Rivest, and Clifford Stein. *Introduction to algorithms, 3rd edition*, chapter 4.5, pages 93 – 96. The MIT Press, Cambridge, Massachusetts, 2009.
35. Intel Corporation. Intel® 64 and ia-32 architectures software developer's manual. 1 – 3, 2011.
36. Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 281–290, New York, NY, USA, 2010. ACM.
37. N. Daswani and M. Stoppelman. The anatomy of clickbot.a. *Proc. of First Workshop on Hot Topics in Understanding Botnets.*

38. Solar Designer. Jpeg com marker processing vulnerability in netscape browsers. *Online: www.openwall.com/advisories/OW-002-netscape-jpeg/*, 2000.

39. CVE details. Security vulnerabilities published in 2013. *Online: http://www.cvedetails.com/vulnerability-list.php*.

40. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52:365–473, May 2005.

41. T. Detristan, T. Ulenspiegel, Y.Malcom, and M.Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack Issue 0x3d*, 2003.

42. C. Eagle. The ida pro book: The unofficial guide to the world's most popular disassembler. 2008.

43. Manuel Egele, Engin Kirda, and Christopher Kruegel. Mitigating drive-by download attacks: Challenges and open problems. In Jan Camenisch and Dogan Kesdogan, editors, *iNetSec 2009 – Open Research Problems in Network Security*, volume 309 of *IFIP Advances in Information and Communication Technology*, pages 52–62. Springer Berlin Heidelberg, 2009.

44. Ifar Erlingsson, Yves Younan, and Frank Piessens. *Handbook of Information and Communication Security*, chapter 30, pages 633 – 658. Springler-Verlag.

45. exploitdb. Atphttpd 0.4 b buffer overflow vulnerabilities // http://www.exploit-db.com/exploits/21614/.

46. exploitdb. Bind 8.2.x (tsig) remote root stack overflow exploit // http://www.exploit-db.com/exploits/280/.

47. exploitdb project. Exploit database. *Online: http://www.exploit-db.com/*.

48. FBI. Cooperation disrupts multi-country cyber theft ring. *Press Release, FBI National Press Office*, October 2010.

49. Éric Filiol. E.: Metamorphism, formal grammars and undecidable code mutation. *J. Comp. Sci*, pages 70–75, 2007.

50. Eric Filiol. Malicious cryptography techniques for unreversable (malicious or not) binaries. *arXiv preprint arXiv:1009.4000*, 2010.

51. Andrea Flexeder, Bogdan Mihaila, Michael Petter, and Helmut Seidl. Interprocedural control flow reconstruction. In Kazunori Ueda, editor, *Programming Languages and Systems*, volume 6461 of *Lecture Notes in Computer Science*, pages 188–203. Springer Berlin Heidelberg, 2010.

52. Inc Free Software Foundation. A gnu manual. *A GNU Manual*, 2014.

53. S. Gaivoronski and D. Gamayunov. Hide and seek: Worms digging at the internet backbones and edges. In *Proceedings of the 7th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2013)*, pages 94–107. National Research Technical University Kazan, Russia: Kazan, 2013.

54. Dennis Gamayunov, Nguyen Thoi Minh Quan, Fedor Sakharov, and Edward Toroshchin. Racewalk: Fast instruction frequency analysis and classification for shellcode detection in network flow. In *Proceedings of the 2009 European Conference on Computer Network Defense*, EC2ND '09, pages 4–12, Washington, DC, USA, 2009. IEEE Computer Society.

55. Gera and Riq. Advances in format string exploiting. *Phrack Magazine*, 59(7), July 2001.

56. TMS470R1x User's Guide. *32-Bit RISC Microcontroller Family*, chapter 5, pages 5–1 – 5–46. Texas Instruments.

57. O. Horovitz. Big loop integer protection. *Phrack Magazine*, 60(9), 2002.

58. J. Huang. Detection of data flow anomaly through program instrumentation. *IEEE Transsaction on Software Engineering*, 5(3), May 1979.

59. L. Hui. Color set size problem with application to string matching. *Proceeding of the 3rd Symposium on Combinatorial Pattern Matching*, 1992.

60. SANS Institute. Lion worm. http://www.sans.org/ y2k/lion.htm.

61. K2. Admmutate. http://www.ktwo.ca/security.html.

62. M. Kaempf. Vudo malloc tricks. *Phrack Magazine*, 57(8), 2001.

63. Daniel Kästner and Stephan Wilhelm. Generic control flow reconstruction from assembly code. *SIGPLAN Not.*, 37(7):46–55, June 2002.

64. Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

65. Johannes Kinder and Dmitry Kravchenko. Alternating control flow reconstruction. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 267–282. Springer Berlin Heidelberg, 2012.

66. Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In NeilD. Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *Lecture Notes in Computer Science*, pages 214–228. Springer Berlin Heidelberg, 2009.

67. Jack Koziol, Dave Aitel, David Litchfield, Chris Anley, Neel Mehta, and Riley Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*, chapter 2, pages 11 – 35. Wiley Publishing, Inc.

68. Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *In RAID*, pages 207–226. Springer-Verlag, 2005.

69. Kirill Kruglov. Monthly malware statistics: June 2010. *Kaspersky Lab Report*, June 2010.

70. Zhichun Li, Manan Sanghi, Yan Chen, Ming yang Kao, and Brian Chavez. Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience. In *S&P'06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 32–47. IEEE Computer Society, 2006.

71. Ziqing Mao, Ninghui Li, and Ian Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *Financial Cryptography and Data Security*, pages 238–255. Springer, 2009.

72. B. McKay. Nauty: No automorphism, yes? *thhp://cs.anu.edu.au/ bdm/nauty/*.

73. B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30, 1981.

74. James Newsome. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 226–241, 2005.

75. Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

76. A. Pasupulati, J. Coit, K. Levitt, J. C. Kuo, and K. P. Fan. Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In *Proceedings of the Network Operations and Management Symposium (NOMS*, pages 235–248, 2004.

77. Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode.

78. Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Network-level polymorphic shellcode detection using emulation. In *In Proceedings of the GI/IEEE SIG SIDAR Conference on Detection*

of Intrusions and Malware and Vulnerability Assessment (DIMVA, pages 54–73, 2006.

79. Phillip Porras, Hassen Saïdi, Vinod Yegneswaran, Phillip Porras, Hassen Saïdi, and Vinod Yegneswaran. A multi-perspective analysis of the storm (peacomm) worm. *Online: http://www.cyber-ta.org/pubs/StormWorm/report*.
80. ARM Project. Arm architecture reference manual. 2005.
81. ARM Project. The arm instruction set. Technical report, ARM University Program, 2010.
82. Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, Nagendra Modadugu, et al. The ghost in the browser analysis of web-based malware. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 4–4, 2007.
83. Rafael Rodríguez-Gómez, Gabriel Maciá-Fernández, and Pedro Garcia-Teodoro. Analysis of botnets through life-cycle. In Javier Lopez and Pierangela Samarati, editors, *SECRYPT*, pages 257–262. SciTePress, 2011.
84. Christian Rossow, Dennis Andriesse, Tillmann Werner, Brett Stone-Gross, Daniel Plohmann, Christian J. Dietrich, and Herbert Bos. P2PWNED: Modeling and Evaluating the Resilience of Peer-to-Peer Botnets . In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)* , San Francisco, CA, May 2013.
85. Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 289–300, Washington, DC, USA, 2006. IEEE Computer Society.
86. Bruce Schneier. *Applied Cryptography (2Nd Ed.): Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

87. Bruce Schneier. *Applied Cryptography (2Nd Ed.): Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

88. Benjamin Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 45–54, 2002.

89. scut team teso. Exploiting format string vulnerabilities. *Online: www.team-teso.net*, 2001.

90. M. Sedalo. Jempiscodes: Polymprphic shellcode generator. *www.shellcode.com.ar/en/proyectos.html*.

91. Multi-State Information Sharing and Analysis Center. Vulnerability in oracle java runtime environment could allow remote code execution. *Online: https://msisac.cisecurity.org/advisories/2013/2013-041.cfm*.

92. T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, (147):195–197, 1981.

93. Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis, and Salvatore J. Stolfo. On the infeasibility of modeling polymorphic shellcode. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 541–551, New York, NY, USA, 2007. ACM.

94. J. Stewart. Bobax trojan analysis. Technical report, SecureWorks, 2004.

95. Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Your botnet is my botnet: Analysis of a botnet takeover. *Technical report, University of California*, May 2009.

96. Matt Sully and Matt Thompson. The deconstruction of the mariposa botnet. In *Defence Intelligence (whitepaper)*, 2010.

97. Microsoft Security TechCenter. Microsoft security bulletin ms12-020 - critical: Vulnerabilities in remote desktop could allow remote code execution (2671387). *Online: http://technet.microsoft.com/en-us/security/bulletin/ms12-020*.

98. H. Theiling. Extracting safe and precise control flow from binaries. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 23–30, 2000.

99. Thomas Toth and Christopher Kruegel. Accurate buffer overflow detection via abstract payload execution. In *In RAID*, pages 274–291, 2002.

100. Lanjia Wang, Hai-Xin Duan, and Xing Li. Dynamic emulation based modeling and detection of polymorphic shellcode at the network level. *Science in China Series F: Information Sciences*, 51(11):1883–1897, 2008.

101. Xinran Wang, Yoon-chan Jhi, Sencun Zhu, and Peng Liu. Protecting web services from remote exploit code: a static analysis approach. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 1139–1140, New York, NY, USA, 2008. ACM.

102. Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu. Sigfree: A signature-free buffer overflow attack blocker. *Ieee Transactions On Dependable And Secure Computing*, 7(1):65–79, 2010.

103. Mark David Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, Ann Arbor, MI, USA, 1979. AAI8007856.

104. Yarom and Yuval. Method of relocating stack in a computer system for preventing overrate by an exploit program. *US patent 5949973, assigned to Memco Software, Ltd.*, 1999.

105. Mark Vincent Yason. The art of unpacking. *In Proc. of BLACKHAT security conference*, 2007.

106. M. Zalewski. Remote vulnerability in ssh daemon crc32 compression attack detector. *Online: www.bindview.com/Support/RAZOR/Advisories/2001/adv_ ssh1crc.cfm*, 2001.

107. Qinghua Zhang, Douglas S. Reeves, Peng Ning, and S. Purushothaman Iyer. Analyzing network traffic to detect selfdecrypting exploit code. In *In Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS*, 2007.

108. К. В. Воронцов. Лекции по логическим алгоритмам классификации. *МГУ*, 2007.

109. С.А. Гайворонская. Методы обнаружения вредоносного исполнимого кода в высокоскоростных каналах передачи данных. *Системы высокой доступности*, 2(7):70–75, 2011.

110. С.А. Гайворонская. Гибридный метод обнаружения шеллкодов. *Системы высокой доступности*, 2(8):33–44, 2012.

111. С.А. Гайворонская и Д.Ю. Гамаюнов. Иерархическая топология декомпозированных алгоритмов для обнаружения вредоносного исполнимого кода. *Материалы XX Международной научной конференции студентов, аспирантов и молодых ученых «Ломоносов 2013» [Электронный ресурс]*, pages 10–13, 2013.

112. С.А. Гайворонская и И.С. Петров. Исследование средств автоматической генерации вредоносного исполнимого кода некоторого класса для мобильных платформ. *Программные системы и инструменты. Тематический сборник (2013)*, 14:72–82, 2013.

Приложение А. Алгоритм восстановления IFG

Алгоритм восстановления графа потока инструкций из вектора дизассемблированных инструкций. На вход алгоритму поступает вектор дизассемблированных цепочек инструкций и длина входного потока. Выходом алгоритма является граф потока инструкций IFG. Высокоуровневое описание алгоритма может быть представлено следующими шагами:

1. Инициализация корневой вершины IFG пустым значением. Все последующие независимые друг от друга подграфы являются потомками корневой вершины.
2. Для каждого глобального смещения входного буфера осуществляется построение подграфов потока инструкций:
3. Для каждого локального смещения входного буфера, начинающегося со значения глобального смещения, осуществляется построения графа потока инструкций следующим образом:
4. Проверить, существует ли в IFG вершина, соответствующая инструкции с анализируемым смещением. Если такая инструкция существует, вернуться на шаг 2. Это означает, что граф потока инструкций, начинающийся с анализируемого смещения, уже восстановлен. Иначе перейти на шаг 5.
5. Извлечь из ВЦДИ инструкцию, соответствующую анализируемому смещению. Если такой инструкции в ВЦДИ не обнаружено, это значит, что с анализируемого смещения поток входных данных не был корректно дизассемблирован. В этом случае вернуться на шаг 2. Иначе перейти на шаг 6.
6. Если инструкция не является инструкцией перехода (как условного, так и безусловного), перейти на шаг 11. Иначе перейти на шаг 7.
7. Операнд инструкции условного перехода является либо абсолютным значением адреса перехода, либо регистром, в котором содержится аб-

солютное значение адреса, либо указанием на память, где так же содержится абсолютное значение адреса перехода. При восстановлении IFG рассматриваются только переходы с абсолютным значением адреса. Так как шаг восстановления графа потока инструкций представляет из себя статический анализ входного потока данных, значения регистров и памяти вычислить невозможно. Для таких инструкций граф далее не восстанавливается.

На данном шаге проверяется наличие в IFG инструкции со смещением `target`. Если такая инструкция в графе обнаружена, перейти на шаг 8, иначе перейти на шаг 9.

8. Восстановить дугу графа потока инструкций, исходящую из текущей инструкции и входящую в инструкцию, расположенную по адресу `target`. Перейти на шаг 10.
9. Добавить инструкцию с адресом `target` в массив внешних инструкций. *Внешней* называется инструкция, по адресу которой осуществляется переход потока управления, но которая еще не была разобрана в процессе восстановления графа потока инструкций. Помимо информации об адресе внешней инструкции, массив внешних инструкций содержит так же информацию об инструкциях, которые осуществляют переход потока управления на нее.
10. Если команда является командой условного перехода, восстановить дугу графа потока инструкций, исходящую из текущей инструкции и входящую в инструкцию, расположенную в памяти за текущей.
11. Проверить, содержится ли анализируемая инструкция в массиве внешний инструкций `ExtArray`. Если анализируемая инструкция найдена в `ExtArray`, восстановить ребра графа потока инструкций, исходящих из инструкций, осуществляющих переход графа потока инструкций на текущую. Вернуться на шаг 3.

Листинг алгоритма восстановления графа потока инструкций на псевдоязыке приведен ниже.

```
1: IFG = null;
2: last_instruction = IFG;
3: for (global_offset = 0; global_offset < buffer_lenght; global_offset++)
4:     for (local_offset = global_offset; local_offset < buffer_lenght;
   local_offset++)
5:             if (IFG.find(local_offset)) continue;
6:             instruction = Flows.Get(local_offset);
7:             if (instruction == null) continue;
8:             makeConnection(last_instruction, instruction);
9:             if (instruction.Type == JMP || instruction.Type == JMPC)
10:                    if (IFG.find(target))
11:                           makeConnection(instruction, target);
12:                    else
13:                           ExtArray.Add(target);
14:                    if (instruction.Type == JMP)
15:                           last_instruction = IFG;
16:                           break;
17:            last_instruction = instruction;
18:            if (ext = ExtArray.Find(instruction))
19:                    forall(parent in ext.parents)
20:                           makeConnection(parent, instruction);
```

Анализ алгоритма.

Будем оценивать алгоритмическую сложность алгоритма для входного потока, состоящего из n байт. В общем случае, количество итераций алгоритма, вызываемых на шагах 3-4, будет равно n^2. Тем не менее, на шаге 5 алгоритма проверяется, производился анализ инструкции с заданным смещением, и в случае положительно ответа, основное тело циклов алгоритма не запускается. Таким образом, инструкции для разных смещений от начала входного потока будут проанализированы ровно один раз,

что делает значение количества повторений тела циклов алгоритма равным n. Проверка, была ли уже проанализирована инструкция с заданного смещения, производится путем анализа наличия флага в массиве, элементы которого соответствуют различным смещениям. Таким образом, сложность шагов 4 и 10 оценивается значением $O(1)$. Сложность поиска инструкции в ВЦДИ оценивается значением $O(lgn)$. Сложность процедуры `makeConnection`$(addr_1, addr_2)$ оценивается значением $O(2lgn)$, так как включает в себя поиск элементов $addr_1, addr_2$ в дереве, представляющем IFG. Сложность поиска элемента в массиве внешних инструкций так же оценивается значением $O(n)$. Там образом, суммарная вычислительная сложность шагов 1-18 приведенного алгоритма можно оценить соотношением

$$Complexity(1-18) \approx O(n(3lgn + C)) = O(nlgn), \tag{A.1}$$

где C - некоторая константа. Несмотря на то, что в худшем случае сложность шагов 19-20 оценивается значением $O(nlgn)$, на практике такая ситуация недостижима. Не зависимо от наличия внешних циклов, тело внутреннего цикла `forall` будет выполнено не более, чем n раз в виду того, что каждая из n инструкций может осуществлять передачу управления не более, чем на одну инструкцию, расположенную не после нее непосредственно. Таким образом, суммарная сложность работы алгоритма представима в виде

$$Complextity(1-20) \approx O(nlgn + nlgn) = O(nlgn). \tag{A.2}$$