ОПРЕДЕЛЕНИЕ ГРАНИЦ ПОДПРОГРАММ ПРИ СТАТИЧЕСКОМ АНАЛИЗЕ БИНАРНЫХ ОБРАЗОВ

Александров Я.А.[1], Сафин Л.К.[2], Чернов А.В.[3], Трошина К.Н.[4]

В процессе исследования защищенности программы по требованиям информационной безопасности представляется целесообразным использование статического анализа бинарного образа программы. Важным этапом статического бинарного анализа является задача разбиения бинарного образа на подпрограммы. Распространенный алгоритм решения данной задачи опирается на определение стартовых адресов подпрограмм, начинающихся с определенной последовательности инструкций машинного кода, и последующего обхода потока управления в глубину от точки входа до инструкций возврата. В работе предлагается алгоритм нахождения границ подпрограмм, основанный на поиске стартовых адресов подпрограмм на основе набора эвристических критериев и последующем рекурсивном обходе потока управления, включающий распознавание некоторых конструкций языков высокого уровня и функций стандартных библиотек.

***Ключевые слова**: статический анализ, бинарный анализ, подпрограмма, граф потока управления, дизассемблер, сигнатурный поиск, таблица виртуальных функций, структуры RTTI, рекурсивный обход, распознавание функций стандартных библиотек.*

1. Введение

В настоящее время в результате интенсивного роста использования программного обеспечения в большинстве областей деятельности человека ощущается нехватка опытных специалистов, разрабатывающих код в соответствии с требованиями информационной безопасности. На помощь разработчикам приходят автоматизированные средства статического анализа исходного кода, которые позволяют проверять программное обеспечение на наличие уязвимостей и ошибок на ранних стадиях разработки. Статический анализ включает в себя анализ потоков управления и данных, семантический анализ и анализ межпроцедурного взаимодействия [1, 2]. На данный момент существует большое количество различных алгоритмов статического анализа, широко использующихся в подобных инструментальных средствах. Статический анализ, в отличие от динамического анализа, осуществляет полное покрытие исходного кода.

При отсутствии исходных кодов востребованными являются средства анализа бинарных образов программного обеспечения. Использование сторонних систем в критических к информационной целостности областях деятельности без должной проверки на соответствие требованиям информационной безопасности может привести к серьезным финансовым потерям. В данной работе рассматривается статический анализ бинарных образов.

В процессе статического бинарного анализа исполняемые файлы переводятся в промежуточное представление, и далее на этом промежуточном представлении применяются стандартные алгоритмы статического анализа. Важнейшим этапом такого анализа является определение границ подпрограмм в бинарном образе. Этой задаче посвящена данная работа. Результаты работы применимы к программам, полученным при трансляции исходных кодов на языках высокого уровня C и C++ в бинарный образ программы для архитектуры микропроцессоров x86.

В архитектуре x86 инструкции не обладают фиксированной длиной. В бинарных образах отсутствует четкое разделение на секции кода и данных: таблицы импорта, таблицы виртуальных функций могут находиться в секции кода, таблицы переходов могут находиться в промежутках между базовыми блоками функций в секции кода. Такие трудности приводят к необходимости создания нетривиальных алгоритмов, решающих поставленную выше задачу.

Наиболее распространены два метода решения задачи определения стартовых адресов под-

1 Александров Ярослав Алексеевич, Московский государственный университет им. М.В. Ломоносова, г. Москва, yaroslavalexandrov@gmail.com.

2 Сафин Ленар Камилевич, Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В.И. Ульянова (Ленина), г. Санкт-Петербург, safin.l91@gmail.com.

3 Чернов Александр Владимирович, кандидат физико-математических наук, Московский государственный университет им. М.В. Ломоносова, г. Москва, blackav@gmail.com.

4 Трошина Катерина Николаевна, кандидат физико-математических наук, Компания SmartDec, г. Москва, katerina@smartdec.ru.

программ в бинарном образе. В первом методе адреса подпрограмм определяются по стандартному прологу (для архитектуры x86 – это **push ebp; mov ebp, esp**). Во втором методе происходит рекурсивный обход секции кода от точки входа с распознаванием инструкций вызова подпрограмм [3]. Обход осуществляется за счет распознавания инструкций перехода. Также применяют комбинации описанных методов, когда рекурсивный обход запускается из найденных по прологу стартовых адресов. Данный алгоритм в таком виде показывает низкий процент распознавания стартовых адресов подпрограмм, что связано с отсутствием указанного выше стандартного пролога у большого количества подпрограмм, наличием косвенных вызовов подпрограмм (например, при вызове виртуальных функций), наличием косвенных переходов (например, косвенные переходы возникают при компилировании конструкции switch).

В данной работе предлагаются методы, которые улучшают описанный выше базовый алгоритм. Анализ тестовой базы исполняемых файлов показал, что большая часть подпрограмм не начинается со стандартного пролога. С помощью автоматического анализа тестовой базы были найдены вариации стандартного пролога, а также новые прологи, с помощью которых можно обнаруживать стартовые адреса подпрограмм. Также в контексте распознавания стартовых адресов подпрограмм предлагается восстанавливать некоторые конструкции языка высокого уровня. Подпрограммы, использующие механизм исключений, обладают фиксированным набором прологов, вызывающих стандартные функции, связанные с обработкой исключений. С помощью распознавания таких функций можно как распознавать стартовые адреса подпрограмм, так и получать информацию о базовых блоках подпрограммы из структур определённого вида. Также предлагается метод распознавания таблиц виртуальных функций, в которых хранятся стартовые адреса подпрограмм. Для верификации полученных с помощью описанных выше методов стартовых адресов подпрограмм предлагается набор эвристических критериев, связанных с распознаванием в секции кода инструкций перехода и инструкций вызова по найденным адресам, а также поиск адресов в секции кода.

После нахождения стартовых адресов подпрограмм возникает задача определения границ каждой подпрограммы. Для решения данной задачи используется рекурсивный обход базовых блоков подпрограммы со стартового адреса с распознаванием инструкций перехода и инструкций возврата. Сложности заключаются в наличии косвенных переходов и выходов из подпрограмм без инструкций возврата (с помощью вызова стандартных no-return функций, таких как exit(), ExitProcess() и т.п.). В контексте распознавания косвенных переходов в работе предложен метод распознавания операторов ветвления switch на ранней стадии анализа бинарного кода, базирующийся на анализе контекста косвенных переходов бинарного приложения. Для распознавания вызовов no-return функций предлагается использовать анализ таблицы импортируемых функций бинарного образа и распознавание подпрограмм-заглушек, которые осуществляют вызов импортируемых функций.

В контексте определения границ подпрограмм, а также анализа семантики бинарного приложения представляется целесообразным распознавание функций стандартных библиотек, таких как стандартная библиотека языка C. В зависимости от используемых конструкций языка (таких, как структурная обработка исключений) и настроек среды разработки, функции стандартных библиотек могут быть встроены в приложение компилятором и/или статическим компоновщиком. Распознавание таких функций возможно с использованием методов поиска по сигнатурам. В данной работе предлагаются методы автоматизированного сбора сигнатур функций статических библиотек. На первом этапе процесса сбора сигнатур отделяются тела функций статической библиотеки. На втором этапе производится анализ собранных сигнатур на предмет наличия в них изменяющихся байтов (например, смещений, зависящих от расположения в бинарном коде).

В указанных выше предлагаемых алгоритмах часто возникает необходимость применения сигнатурного поиска. В работе предложен алгоритм сигнатурного поиска, являющий обобщением классического алгоритма поиска подстрок Ахо-Корасик [4].

При тестировании разработанных в рамках данной работы алгоритмов использовалась база бинарных образов, состоящая из 1052 исполняемых файлов, полученных из программ на языках C и C++ с помощью компиляторов MSVC и GCC разных версий. При определении стартовых адресов подпрограмм определялись два критерия качества реализуемого алгоритма. Первый заключался в проценте правильно распознанных адресов (процент от общего количества стартовых адресов подпрограмм), второй – в проценте неправильно распознанных адресов (процент

от общего количества найденных адресов). Для сравнения использовались данные, полученные из анализа тестовой базы с помощью пакетного запуска интерактивного дизассемблера IDA Pro [5] с выполнением соответствующих скриптов. Описанный выше стандартный алгоритм определения стартовых адресов подпрограмм показывал 22% верных нахождений и 0% неверных нахождений.

Далее в разделе 2 будут описаны предлагаемые методы определения стартовых адресов подпрограмм: анализ прологов, таблиц виртуальных функций, эвристические критерии определения стартовых адресов. В разделе 3 будет описан метод определения границ подпрограмм: базовый метод, метод распознавания конструкции switch, связанные с определением границ эвристики. В разделе 4 описывается метод распознавания функций стандартных библиотек. В разделе 5 будет описан предлагаемый метод сигнатурного поиска. Раздел 6 является заключением данной работы.

2. Определение стартовых адресов подпрограмм

Предлагаемый в данной работе алгоритм определения стартовых адресов подпрограмм состоит из двух шагов. На первом шаге определяется набор адресов, которые могут быть стартовыми адресами подпрограмм (с помощью анализа прологов и таблиц виртуальных функций). На втором шаге происходит рекурсивный обход подпрограмм из найденных на первом шаге адресов с распознаванием инструкций вызова подпрограмм.

Анализ тестовой базы исполняемых файлов показывает, что значительная часть подпрограмм не использует стандартный пролог **push ebp; mov ebp, esp**, а использует либо его модификацию, либо другие прологи. В рамках данной работы проведен автоматизированный сбор прологов подпрограмм. Для каждого количества инструкций от одного до определённого предельного числа (в данной работе число инструкций в прологе не превышало пяти) ищется частота появления пролога, состоящего из данного количества инструкций, в подпрограммах тестовой базы бинарных образов. В рамках данной работы такой автоматизированный сбор осуществлялся с помощью пакетного запуска интерактивного дизассемблера IDA Pro с выполнением скрипта сбора прологов. Анализ тестовой базы выявил некоторые вариации стандартного пролога, а также новые прологи, широко использующиеся в бинарных образах:

1. **mov edi, edi** (hot patch prologue);
2. **push ebp; xor eax, eax; mov ebp, esp;**
3. **push esi; mov esi, ecx**
4. **push ebp; push edi; push esi; push ebx; sub esp, …**

Простой поиск по указанным выше прологам может приводить к появлению «ложных срабатываний». Для уменьшения количества таких ошибок найденные адреса верифицируются с помощью одного из двух правил:

Данный адрес встречается в машинном коде бинарного образа;

В секции кода бинарного образа существует инструкция вызова или инструкция перехода по данному адресу.

Помимо описанных выше прологов, подпрограммы, использующие механизм обработки исключений, также используют фиксированный набор прологов:

1. **push … ; push offset scopetable_addr; call _SEH_prolog**
2. **push … ; push offset scopetable_addr; call _SEH4_prolog**
3. **mov eax, offset label; call _EH_prolog**

Данные инструкции свойственны компилятору MSVC на платформе Windows [6]. В данных прологах **scopetable_addr** является адресом структуры специального вида, в которой хранится информация, связанная с обработкой исключений (например, адрес базового блока-обработчика исключения). С помощью адреса **label** также можно получить адрес такой структуры. Функции **_SEH_prolog**, **_SEH4_prolog** (для языка C) и **_EH_prolog** (для C++) являются функциями стандартного вида в точности до адресов некоторых вызываемых вспомогательных функций. Распознавание стартовых адресов подпрограмм, использующих механизм обработки исключений, необходимо проводить в два этапа. На первом этапе происходит распознавание функций **_SEH_prolog**, **_SEH4_prolog** и **_EH_prolog** в секции кода. Далее на втором этапе по определённому выше виду прологов происходит распознавание стартовых адресов таких подпрограмм.

В бинарных образах, полученных из исходного кода на языке C++, в котором используются виртуальные функции, код подпрограмм, полученных из этих виртуальных функций, часто не включает в себя стандартные прологи. Вызов таких подпрограмм не происходит явным образом. Эти факторы затрудняют распознавание таких подпрограмм. В данной работе для распознавания таких подпрограмм осуществляется поиск таблиц виртуальных функций в бинарном образе. Таблица

Таблица 1. *Изменение качества работы алгоритма*

Алгоритм	Процент верно найденных адресов	Процент неверно найденных адресов
Базовый алгоритм	22%	0%
Расширение базы стандартных прологов без верификации / с верификацией адресов	63%	3% / 1%
Добавление прологов, связанных с обработкой исключений	75%	1%
Распознавание таблиц виртуальных функций	86%	1%
Распознавание инструкции вызова подпрограмм в процессе рекурсивного обхода графа потока управления	87%	1%
Распознавание конструкций switch	88%	1%

виртуальных функций представляет собой список двойных слов – адресов виртуальных функций. Для поиска таблиц виртуальных функций используется алгоритм [7].

В результате работы первого шага предлагаемого алгоритма формируется множество стартовых адресов, состоящих из верифицированных адресов описанных в данной работе прологов, адресов, найденных по прологам, использующим механизм исключений, и адресов, полученных из таблиц виртуальных функций.

На втором шаге алгоритма для всех адресов из найденного на первом шаге множества осуществляется рекурсивный проход декодирующим модулем от данного адреса до инструкции перехода или инструкции возврата; если встречается инструкция перехода, то происходит аналогичный рекурсивный проход от адресов, на которые ссылается данная инструкция перехода. В процессе такого обхода при декодировании инструкций вызова подпрограмм их целевые адреса добавляются в искомое множество стартовых адресов подпрограмм, при этом от данных адресов также осуществляется аналогичный рекурсивный обход. При обработке инструкций перехода применяется алгоритм распознавания конструкции switch, описанный в разделе 3. Таким образом, найденное на первом шаге множество адресов дополняется распознанными на втором шаге в результате описанного выше рекурсивного обхода целевыми адресами функций вызова подпрограмм.

На втором шаге происходит декодирование инструкций, которые встречаются при рекурсивном обходе, а значит всех инструкций, которые составляют тела найденных подпрограмм. С помощью этого множества инструкций осуществляется второй этап верификации адресов, найденных по стандартным прологам подпрограмм. Для каждого верифицированного на первом шаге адреса проверяется, был ли он верифицирован только инструкциями перехода. Если это так, и ни одна из инструкций перехода не встречается в полученном на втором шаге множестве инструкций, то данные адрес удаляется из множества стартовых адресов подпрограмм. Проверка именно инструкций перехода, но не вызова подпрограмм, связана с полученным эмпирическим путём количеством ложных срабатываний в обоих случаях.

Поиск прологов, функций _SEH_prolog, __SEH4_prolog и _EH_prolog, инструкций **mov** и **push** при распознавании таблиц виртуальных функций, верификация адресов по данным осуществляются с помощью сигнатурного поиска по бинарному образу. Предлагаемый в данной работе метод сигнатурного поиска описан в разделе 5.

В таблице 1 показано изменение качества распознавания стартовых адресов подпрограмм бинарного образа в процессе добавления описанных выше эвристик. В первом столбце указаны добавляемые к базовому алгоритму эвристики, во втором и третьем – процент верно и неверно распознанных стартовых адресов подпрограмм соответственно. На данной тестовой базе предложенный в работе алгоритм показывает качество 88% и 1% соответственно. При этом в 42% случаях распознавание стартовых адресов подпрограмм прошло абсолютно верно (качество 100% и 0% соответственно).

Полученный процент верно найденных адресов подпрограмм можно объяснить двумя способами:

1. IDA Pro допускает ложные срабатывания: в результате ручного анализа были найдены случаи распознавания кода на месте, где хранятся данные (например, строковые);

2. Некоторые подпрограммы могут не обладать стандартными прологами, при этом не вызываться из кода бинарного образа.

3. Определение границ подпрограмм

После определения стартовых адресов подпрограмм бинарного образа необходимо обозначить границы каждой подпрограммы. При этом подпрограммы могут не занимать непрерывную область образа, а могут как включать области данных (например, таблицы переходов), так и другие подпрограммы. Для верного определения области, которую занимает подпрограмма в памяти, в данной работе предлагается проводить рекурсивный обход подпрограммы, начиная с её стартового адреса, аналогичный обходу, описанному в разделе 2 на втором шаге алгоритма определения стартовых адресов подпрограмм.

В процессе обхода важно обратить внимание, что помимо инструкций возврата подпрограммы могут завершаться вызовами так называемых no-return функций (функций, не возвращающих управления). Для более точного определения границ таких подпрограмм осуществляется итерационное наращивание множества no-return подпрограмм. Изначально множество инициализируется импортируемыми no-return функциями и подпрограммами-заглушками, единственной функциональностью которых является вызов импортируемых no-return функций. Далее, если в процессе работы алгоритма определения границ подпрограмм находится новая no-return подпрограмма, то множество наращивается, а алгоритм запускается заново для всех функций.

Как и в случае обхода при поиске стартовых адресов подпрограмм, при поиске границ подпрограмм осуществляется распознавание конструкций switch. Бинарный код, генерируемый транслятором исходного кода языка программирования C для управляющей конструкции switch, зависит от контекста, в котором осуществляется переход по значению операнда.

1. Последовательное сравнение значения операнда с возможными значениями. Значение операнда последовательно сравнивается с каждым из значений case-блоков и, в случае если значения совпадают, осуществляется условных переход.

2. Косвенный переход по таблице диспетчеризации. Компилятор создает таблицу переходов для каждого из значений операнда управляющей конструкции switch. Переход осуществляется командой косвенного перехода **jmp ds:offset[reg*PTR_SIZE]**, где **offset** – адрес таблицы переходов, **reg** – имя регистра, содержащего значение операнда, а **PTR_SIZE** – размер адреса в байтах.

3. Косвенный переход с отображением на таблицу диспетчеризации. Компилятор создает таблицу переходов для ветвей оператора ветвления switch и отображает значение операнда на нее.

Типичные конструкции, используемые современными компиляторами при генерации кода для управляющих конструкций switch, могут быть распознаны с использованием алгоритма простейшего анализа потока данных на основе шаблонов. Идея алгоритма заключается в анализе операнда перехода и восстановлении контекста, в котором осуществляется переход. Для распознавания управляющих конструкций switch, анализ контекста перехода сводится к распознаванию типичных конструкций перехода.

Таблица 2. *Примеры типичных конструкций реализации управляющего оператора switch*

Реализация оператора switch	Поток управления	Процент распознаваемых конструкций switch
Последовательное сравнение значения операнда с возможными значениями	Оператор switch такого типа распознается базовым алгоритмом рекурсивного обхода бинарного кода, описанного в разделе 2.	–
Косвенный переход по таблице диспетчеризации	cmp reg, imm ; imm + 1 – размер таблицы переходов ja/jbe offset1 ; offset1 – смещение адреса базового блока случая по умолчанию jmp ds:offset2[reg * PTR_SIZE] ; offset2 – смещение адреса таблицы переходов	85%

Косвенный переход с отображением на таблицу диспетчеризации	cmp reg, imm ; imm + 1 – размер таблицы переходов
	ja/jbe offset1 ; offset1 – смещение адреса базового блока случая по умолчанию
	movzx reg, ds:offset2[reg] ; offset2 – смещение адреса таблицы отображения
	jmp ds:offset3[reg * PTR_SIZE] ; offset3 – смещение адреса таблицы переходов
		96%

В таблице 2 приводятся примеры типичных конструкций, используемых компиляторами при трансляции оператора ветвления switch. В первом столбце указаны методы реализации оператора switch в бинарном коде. Во втором столбце приведены примеры шаблонов потока инструкций, используемые современными компиляторами языков программирования C и C++. В третьем столбце указан процент правильно распознанных конструкций switch от общего числа используемых конструкций switch в выборке бинарных образов программ на языках программирования C и C++.

4. Распознавание функций стандартных библиотек

Распознавание функций стандартных библиотек является необходимым шагом в процессе семантического анализа и анализа потока данных в контексте анализа потенциальных уязвимостей программного кода, в частности, таких как неправильная работа с памятью или потоками входных и выходных данных. В зависимости от настроек программного окружения среды компилятора, такие функции могут быть встроены в бинарный образ программы. В таком случае для распознавания функций стандартных библиотек могут быть применены методы сигнатурного поиска.

В процессе компоновки бинарного образа, компоновщик транслирует объектные модули исходной программы с библиотекой стандартных функций. Статический компоновщик строит таблицы символов объектных файлов, разрешает внешние ссылки и встраивает символы в бинарный образ программы. При разрешении внешних имен, компоновщик использует словарь перераспределения (relocation table, fix-up information), заменяя перемещаемые байты в подпрограммах, после чего включает объектный модуль в образ исполняемого файла.

На основе анализа статических библиотек стандартных функций и таблиц перемещаемых байтов могут быть построены сигнатуры стандартных функций. В бинарный образ статических библиотек включаются объектные модули, встраиваемые компоновщиком в образ исполняемого файла. При создании статических библиотек обычно используется файловый формат AR. Каждый объектный модуль статической библиотеки описывается именем модуля, датой последнего изменения, размером и другими служебными данными.

В бинарный образ объектных модулей включается таблица символов, на основе анализа которой может быть извлечена информация об определяемых подпрограммах, адресах подпрограмм и размерах бинарного кода подпрограмм. После определения границ подпрограмм, на их бинарные образы отображается словарь перераспределяемых байтов. Перемещаемые байты отмечаются в сигнатуре подпрограммы как изменяемые и их значения не учитываются при сигнатурном поиске.

К базе сигнатур подпрограмм, составленных при анализе статических библиотек стандартных функций, добавляется информация о внешних именах, используемых подпрограммой. Информация об используемых подпрограммой символах необходима для разрешения неоднозначностей распознавания подпрограмм, сигнатуры которых совпадают. Сигнатуры таких функций неразличимы на этапе сигнатурного поиска, однако, функции могут быть успешно распознаны на этапе анализа внешних вызовов.

Тем не менее, существуют функции, распознавание которых в рамках статического анализа кода представляется затруднительным. Например, исполняемый код некоторых функций стандартной библиотеки C++ полностью совпадает; некоторые подпрограммы, связанные с обработкой исключений и информацией о типах встраиваются в бинарный образ программы более одного раза. В контексте статического анализа бинарного кода неверное распознавание функций стандартных библиотек является нежелательным, а потому неоднозначности распознавания подпрограмм, неразрешенные на этапе анализа внешних вызовов должны быть сведены к минимально возможному значению. Исходя из этого, из базы данных сигнатур подпрограмм исключаются.

1. Сигнатуры подпрограмм, размер бинарного кода которых меньше порогового значения. Размер минимальной сигнатуры определяется на основе анализа кодовой базы программ целевой платформы.

2. Сигнатуры подпрограмм, бинарный код которых полностью совпадает.

3. Сигнатуры подпрограмм, неотличимые в контексте сигнатурного поиска по текущей базы сигнатур, включая этап разрешения внешних ссылок.

К известным реализациям методов распознавания функций стандартных библиотек, следует отнести библиотеку FLIRT (Fast Library Identification and Recognition Technology) [8], реализованную в дизассемблере IDA Pro. При ручном анализе результатов работы поиска функций стандартных библиотек в авторской реализации и в реализации FLIRT были получены следующие результаты.

1. Множество распознанных функций в авторской реализации и множества распознанных функций в реализации FLIRT не совпадают и не содержат друг друга. Пересечение множеств распознанных функций составляет 90% от общего числа встроенных функций стандартных библиотек.

2. Множество распознанных функций в реализации FLIRT составило 94%.

3. Множество распознанных функций в авторской реализации составило 96%. В отличие от FLIRT, в авторской реализации были распознаны функции printf и wprintf.

4. Неверно распознанных функций стандартных библиотек не было обнаружено для обеих реализаций.

В отличие от существующих реализаций, построение базы данных сигнатур подпрограмм производится в автоматическом режиме для любой статической библиотеки, что позволяет инструментальным средствам статического анализа двоичного кода создавать базы данных сигнатур при анализе статических библиотек, которые в дальнейшем могут быть использованы при анализе исполняемых файлов, скомпонованных с такими библиотеками.

5. Метод сигнатурного поиска

При распознавании типичных конструкций бинарного кода используется метод сигнатурного поиска по маске. Сигнатура представляет собой строку байтов, часть из которых отмечена как изменяющиеся. Семантика изменяющегося байта равносильна обозначению любого символа «.», использующегося в регулярных выражениях. Например при поиске байтовой подстроки 74 ?? 74 8b в байтовой строке 73 74 0a 74 8b 3c 8b, будут найдены вхождения в позициях 2 и 4.

Алгоритм сигнатурного поиска основан на классическом алгоритме поиска набора подстрок в строке Ахо-Корасик. Алгоритм описывают следующие шаги.

Сигнатура разбивается на подстроки, не содержащие изменяющихся байтов. Для каждой подстроки сохраняется ее позиция в исходной строке сигнатуры.

Для каждой подстроки создается буфер состояния поиска, каждый элемент которого описывается парой **(cat_num, count)**. Элементы буфера инициализируются нулями.

Для подстрок сигнатур строится конечный недетерминированный автомат по алгоритму Ахо-Корасик.

Исходная строка байтов обходится по алгоритму симуляции работы автомата Ахо-Корасик. Для каждой найденной подстроки сигнатуры производятся следующие действия.

По позиции подстроки в строке сигнатуры определяется предполагаемое начало строки сигнатуры. Для предполагаемой позиции **pos** сигнатуры размера **size** вычисляется значение пары **(curr_cat_num, buf_pos) = pos DIV size.**

По значению **buf_pos** находится элемент буфера состояния поиска **(cat_num, count)**. Если значение **cat_num** совпадает со значением curr_cat_num, значение count увеличивается на единицу, в противном случае значению **cat_num** присваивается значение **curr_cat_num**, а значению **count** присваивается единица.

Если значение **count** совпадает с количеством подстрок исходной строки сигнатуры, а значение **pos + size** не превышает размера исходной строки байт, по которой осуществляется поиск, найдено вхождение сигнатуры в позиции **pos**.

Вычислительная сложность алгоритма совпадает с вычислительной сложностью алгоритма Ахо-Корасик для набора подстрок, не содержащих изменяющихся байтов. Для увеличения производительности, размер буфера может быть увеличен до ближайшего числа, являющегося степенью числа 2, а операция целочисленного деления с остатком должна быть заменена на операции беззнакового сдвига вправо и побитовой конъюнкции по битовой маске.

6. Заключение

В данной работе предложен алгоритм определения границ подпрограмм в бинарном образе. Важность данной задачи обусловлена её использованием в статическом анализе бинарных образов. Алгоритм получается путём добавления к известному базовому алгоритму набора методов и эвристик: расширения базы стандартных прологов, распознаванием конструкция языка высокого уровня (таких как конструкций switch, обработки исключений, таблицы виртуальных функций).

Эксперименты показывают существенный рост качества определения подпрограмм в бинарном образе по сравнению с базовым алгоритмом. В качестве дальнейших направлений исследования можно увеличивать процент распознанных подпрограмм и уменьшать процент неверно найденных подпрограмм с помощью структурного анализа полученных областей подпрограмм: их взаимного расположения и вложенности.

Литература:

1. Steven S. Muchnick. Advanced compiler design implementation. // Morgan Kaufmann, 1997.
2. Muchnick S. S. Program flow analysis: Theory and applications. // Englewood Cliffs, New Jersey : Prentice-Hall, 1981. Vol. 196.
3. Schwarz B., Debray S., Andrews G. Disassembly of executable code revisited //Reverse engineering, 2002. Proceedings. Ninth working conference on. IEEE, 2002. P. 45-54.
4. Aho A. V., Corasick M. J. Efficient string matching: an aid to bibliographic search //Communications of the ACM. 1975. Vol. 18. N 6. P. 333-340.
5. IDA Pro: About [Электронный ресурс]. URL: http://www.hex-rays.com/products/ida/ (дата обращения 29.07.2015).
6. Igor Skochinsky, Reversing Microsoft Visual C++ Part I: Exception Handling [Электронный ресурс]. URL: http://www.openrce.org/articles/full_view/21 (дата обращения 29.07.2015).
7. Igor Skochinsky, Reversing Microsoft Visual C++ Part II: Classes, Methods and RTTI [Электронный ресурс]. URL: http://www.openrce.org/articles/full_view/23 (дата обращения 29.07.2015).
8. IDA F.L.I.R.T Technology: Overview [Электронный ресурс]. URL: http://www.hex-rays.com/products/ida/tech/flirt/index.shtml (дата обращения 29.07.2015).

SUBROUTINES BOUNDS RECOGNITION IN STATIC ANALYSIS OF BINARY IMAGES

Alexandrov Ya.A.[5], Safin L.K.[6], Chernov A.V.[7], Troshina K.N.[8]

Static program analysis of executable (binary image) is useful for information security assessment. An important early stage of binary static analysis is subroutines boundaries recognition. A widely used algorithm for solving this problem is based on finding certain sequences of bytes constituting subroutine prologue instructions and subsequent depth–first traversal of the subroutine from the entry point, terminating at return instructions. In this work we propose an algorithm using additional heuristic criteria for subroutine entry point detection and subsequent depth–first traversal improved by recognition of certain high–level language constructs and library functions.

Keywords*: static analysis, binary analysis, subroutine, control flow graph, disassembler, pattern matching, virtual functions table, RTTI structures, recursive traversal, standard library functions recognition*

References:

1. Steven S. Muchnick. Advanced compiler design implementation. Morgan Kaufmann, 1997.
2. Muchnick S. S. Program flow analysis: Theory and applications. Englewood Cliffs, New Jersey : Prentice-Hall, 1981. Vol. 196.
3. Schwarz B., Debray S., Andrews G. Disassembly of executable code revisited, Reverse engineering, 2002. Proceedings. Ninth working conference on. IEEE, 2002. P. 45-54.
4. Aho A. V., Corasick M. J. Efficient string matching: an aid to bibliographic search, Communications of the ACM. 1975. Vol. 18. N 6. P. 333-340.
5. IDA Pro: About [Electronic resource]. URL: http://www.hex-rays.com/products/ida/ (date accessed 07/29/2015).
6. Igor Skochinsky, Reversing Microsoft Visual C++ Part I: Exception Handling [Electronic resource pc]. URL: http://www.openrce.org/articles/full_view/21 (date accessed 07/29/2015).
7. Igor Skochinsky, Reversing Microsoft Visual C++ Part II: Classes, Methods and RTTI [Electronic resource]. URL: http://www.openrce.org/articles/full_view/23 (date accessed 07/29/2015).
8. IDA F.L.I.R.T Technology: Overview [Electronic resource]. URL: http://www.hex-rays.com/products/ida/tech/flirt/index.shtml (date accessed 07/29/2015).

5 Yaroslav Alexandrov, Lomonosov Moscow State University, Moscow, yaroslavalexandrov@gmail.com;
6 Lenar Safin, Saint Petersburg Electrotechnical University LETI, Saint Petersburg, safin.l91@gmail.com;
7 Alexander Chernov, Ph. D., Lomonosov Moscow State University, Moscow, blackav@gmail.com;
8 Katerina Troshina, Ph. D., SmartDec Company, Moscow, katerina@smartdec.ru.