УДК 004.056.5 DOI:10.31854/1813-324X-2020-6-2-104-112

Идентификация архитектуры процессора выполняемого кода на базе машинного обучения. Часть 2. Способ идентификации

М.В. Буйневич[1, 2], **К.Е. Израилов**[1, 3*]

[1]Санкт-Петербургский государственный университет телекоммуникаций им. проф. М.А. Бонч-Бруевича, Санкт-Петербург, 193232, Российская Федерация
[2]Санкт-Петербургский университет государственной противопожарной службы МЧС России, Санкт-Петербург, 196105, Российская Федерация
[3]Санкт-Петербургский институт информатики и автоматизации Российской академии наук, Санкт-Петербург, 199178, Российская Федерация
*Адрес для переписки: konstantin.izrailov@mail.ru

Информация о статье
Поступила в редакцию 01.06.2020
Принята к публикации 24.06.2020

Ссылка для цитирования: Буйневич М.В., Израилов К.Е. Идентификация архитектуры процессора выполняемого кода на базе машинного обучения. Часть 2. Способ идентификации // Труды учебных заведений связи. 2020. Т. 6. № 2. С. 104–112. DOI:10.31854/1813-324X-2020-6-2-104-112

Аннотация: *Изложены результаты исследования способа идентификации архитектуры процессора исполняемого кода на базе машинного обучения. Во второй части цикла статей производится синтез трехэтапной схемы способа и соответствующего программного средства. Описывается функциональный и информационный слой архитектуры последнего, а также режимы его работы. Осуществляется базовое тестирование средства и приводятся результаты его работы. На примере идентификации файлов с машинным кодом различных архитектур обосновывается работоспособность предлагаемых способа и программного средства.*

Ключевые слова: *информационная безопасность, машинный код, архитектура процессора, машинное обучение, частотно-байтовая модель, сигнатура кода, способ идентификации процессора, программное средство.*

Введение

Одной из важнейших задач в области информационной безопасности является анализ программно-аппаратных элементов информационной системы, включающих файлы, оборудование, встроенный код и т. п. [1]. Исходя из разнородности таких элементов, применяются специализированные способы анализа, «заточенные» под свой объект. Для повышения оперативности и ресурсоэкономности процесса производится предварительное исследование информационной системы в интересах определения ее состава, формирования приоритетного вектора анализа, выбора соответующих средств, сбора метрик и статистик [2]. В большинстве случаев наибольшую, с точки зрения «враждебности», роль играют выполняемые файлы, которые имеют вид машинного кода (далее – МК). Исходя из того, что МК считается «языком процессора», а также большого количества реально применяемых архитектур процессоров (более 50), возникает необходимость в знании такой архитектуры для каждого файла. Это позволит, как в автоматическом режиме произвести анализ файлов предназначенным для этого средством (например, сканером уязвимостей [3], работающим с Intel-набором команд), так и установить количественно-качественный состав информационной системы (архитектуры МК файлов и их соотношение в программной части, коррелирующее и с аппаратной частью). Таким образом, *идентификация архитектуры выполняемого кода* является актуальной задачей в области информационной безопасности.

В первой части цикла статей [4] был произведен анализ предметной области, в результате чего была получена модель, послужившая для доказательства гипотезы, непосредственно примененной при создании схемы идентификации архитектуры процессора. Принцип идентификации (имеющий аналитический вид) состоит в том, что для нее применяется частотно-байтовая модель МК файла, характеризующая частоту появления байт с определенным значением – *сигнатуру* (массив из 256 элементов со значениями от 0 до 1 включительно). Каждый же из МК файлов с определенным ти-

пом архитектуры имеет собственную «картину» распределения (объясняемую целым рядом объективных причин), что как раз и позволяет производить необходимое различение среди них.

Продолжая следовать канонической схеме исследования «анализ → синтез → оценка», в данной статье будет произведен синтез способа идентификации, а также предложена архитектура и описана реализация конкретного программного средства.

1. Синтез способа идентификации

Схема способа идентификации в формальном (аналитическом) виде была описана в первой части цикла. Суть схемы заключается в построении моделей для каждой из процессорных архитектур, а затем корреспондирования с этими моделями исследуемого файла с целью нахождения вероятностей его отнесения к одному из классов МК (т. е. одной архитектуры). Класс с наибольшей вероятностью и будет идентифицируемым; в случае же близости вероятностей нескольких классов можно говорить о гетерогенности наборов инструкций МК, что потребует дополнительных усилий по идентификации.

Для синтеза нового способа идентификации необходимо решить методологическую задачу выбора собственно механизма такого создания. Существуют различные механизмы, наиболее очевидными из которых являются следующие: расширение/специализация, когда частный способ развивается/сужается до решения боле крупной/мелкой задачи; объединение, когда берутся близкие к задаче способы, наиболее необходимые части которых затем соединяются; эвристический, основанный на интуиции («гениальной догадке») исследователя [5]; эмпирический, строящийся на наблюдаемых данных.

Исходя из специфики решаемой задачи, в том числе упоминаемой в первой части цикла, наиболее подходящим механизмом будет объединение специализации (поскольку, как задача обработки бинарных файлов, так и задача классификации имеют апробированные подходы к решению) и эмпирического (по причине наличия множества файлов с МК, сигнатуру которых практически невозможно вычислить вручную).

Следуя такому механизму, способ может быть построен на базе искусственного интеллекта в части одного из наиболее близких «по духу» метода машинного обучения, а именно – классификации [6]. Согласно последнему, способ идентификации может представлять собой обучение его внутренней модели по прецедентам в виде пары: «усредненное частотно-байтовое распределение МК файлов» vs «их принадлежность к определенному типу архитектуры». Набрав подобным образом достаточную «массу» обучаемых данных, модель будет использоваться способом для отображения МК новых файлов к соответствующему классу МК. Естественно, отдельно стоящей задачей является получение обучающей выборки – набора распределений МК и его архитектуры в достаточном количестве, что также должно быть учтено в способе.

Используя предложенную формальную схему способа и выбранный механизм создания, произведем его алгоритмический синтез в виде последовательности шагов. Такое представление позволит впоследствии создать программное средство идентификации. В качестве данных для обучения возьмем сборку Debian для различных архитектур, использованную ранее для получения сигнатур МК [7]. Также учтем тот факт, что выполняемые файлы могут быть двух форматов: PE и ELF, а помимо секции с МК, в файле присутствуют и другие секции с данными, не содержащими инструкции процессора.

Поскольку идентификация должна не только сказать, принадлежит ли МК тестируемого файла известной архитектуре, но и указать на эту архитектуру, то с точки зрения машинного обучения необходимо решить задачу многоклассовой классификации. По причине того, что процесс обработки множества файлов для получения сигнатуры класса МК для каждого процессора занимает заведомо большее время, чем обучение и идентификация одного файла, и при этом выполняется лишь один раз, то целесообразно выполнить этот процесс единожды с сохранением результатов работы (т. е. частотно-байтового распределения инструкций). Для учета приведенных особенностей способ целесообразно поделить на 3 этапа с возможностью их раздельного выполнения (что будет особенно оправдано при сборе сигнатур МК для большого количества файлов). Так, на первом этапе (Этап 1) требуется распаковать файлы с МК для различных архитектур, взятых из образов дистрибутива Debian, во временные директории. На Этапе 2 необходимо построить сигнатуры и сохранить их в промежуточные файлы. На Этапе 3 требуется провести саму идентификацию путем сначала обучения внутренней модели для многоклассовой классификации, а затем тестирования на ней (т. е. нахождения вероятностей отнесения МК к архитектурам процессоров) подаваемых на вход файлов.

Целесообразно иметь возможность идентификации как полноценных выполняемых файлов (в виде заголовков с набором секций), так и «сырого» МК (т. е. последовательности инструкций для выполнения процессором без дополнительной метаинформации). Разделение обучения и тестирования на подэтапы не имеет практического смысла, поскольку время обучения для набора сигнатур занимает предельно короткое время – сигнатуры состоят из 256 байт (в терминах машинного обучения – признаки), а общее количество процессорных архитектур вряд ли превысит 100 (в терминах машинного обучения – обучающие образцы или прецеденты). Схема работы такого способа представлена на рисунке 1.

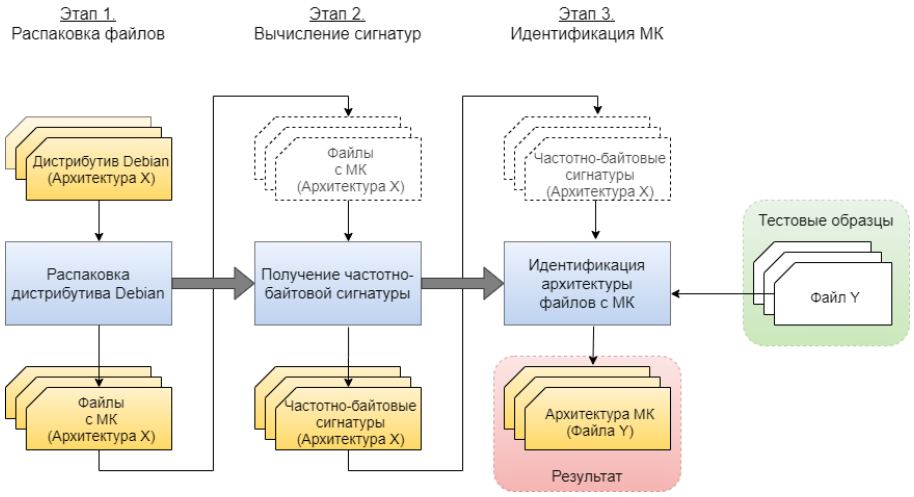

Рис. 1. Схема способа идентификации архитектуры процессора выполняемого кода

Fig. 1. A Scheme of a Method for Identifying a Processor Architecture of Executable Code

Схема состоит из следующих поэтапно выполняемых блоков (принимающих/возвращающих):

Блок 1 – «Распаковка дистрибутива Debian», принимающий на вход дистрибутив под определенную процессорную архитектуру и возвращающий на выходе файлы с МК;

Блок 2 – «Получение частотно-байтовой сигнатуры», принимающий на вход полученные ранее файлы и возвращающий на выходе соответствующую сигнатуру для множества их МК;

Блок 3 – «Идентификация файлов с МК», принимающий на вход все полученные ранее сигнатуры, а также тестируемые файлы, и возвращающий на выходе архитектуры файлов.

В интересах реализации способа необходимо отметить особенности строения дистрибутива Debian. Файлы дистрибутива содержат архивы, имеющие расширение «*.deb» и запакованные алгоритмом 7-Zip. В последних также могут содержаться 7-Zip архивы с расширением «*.tar». Таким образом, для получения всех файлов дистрибутива необходимо распаковать указанные архивы (в том числе вложенные). Исходя из вышеотмеченных особенностей, можно выдвинуть к реализации способа следующие требования:

– по результативности: корректные файлы с МК (т. е. без разрушений и предназначенные для одной процессорной архитектуры) должны однозначно идентифицироваться;

– по оперативности: должна обеспечиваться высокая скорость работы, поскольку в ином случае способ будет не применим для крупных информационных систем (что является вполне реальной ситуацией);

– по ресурсоэкономности: как затраты на программные ресурсы, так и на экспертов, должны быть минимальны.

Соответствие программного средства указанным требованиям может быть оценено непосредственно в процессе его тестирования, что будет осуществлено в заключительной части цикла.

2. Программная реализация средства идентификации

Для разработки программного средства была выбрана среда Microsoft Visual Studio (версии 2019), являющаяся одной из бесспорно ведущих в области IT-инженерии. В том числе и по этой причине языком разработки являлся C# (заведомо поддерживаемый в среде).

Несмотря на высокую и постоянно растущую популярность языка Python [8], в особенности при разработке программ с использованием машинного обучения, он не был применен по целому ряду следующих субъективно негативных признаков: ориентирование на простоту написания кода, что для программистов с большим опытом скорее усложняет разработку, поскольку требует запоминания новых (и не всегда логичных) «простых» конструкций; «родственная близость» C# и Microsoft Visual Studio, что естественно отражается и на использовании первого во втором (поддержка рефакторинга, удобство отладки и т. п.); негативный авторский опыт использования скриптовых языков (Python, Ruby и др.) против построенных на «чистом» байт-коде (C#, Java), а также результаты независимых исследований (к сожалению, не опубликованные в ведущих журналах), которые показывают бóльшую скорость работы и меньшее потребление ресурсов второй группы языков по сравнению с первой, что также повлияло на итоговый выбор.

Даже принимая во внимание библиотеки для машинного обучения ML.Net от компании Microsoft, для разработки данного программного средства применялись сторонние библиотеки Accord.Net, поскольку, как показала практика разработки предыдущих проектов, API от Accord.Net (как и общие шаблоны) оказался более удобной [9].

В качестве многоклассового классификатора был выбран хорошо зарекомендовавший себя алгоритм SVM (*от англ.* Support Vector Machine). И

хотя он предназначен для бинарной классификации (т. е. разделения на два класса), однако его можно адаптировать и для задачи нескольких архитектур путем применения стратегий «один-против-одного» (исходя из суммы голосов каждого попарного классификатора) и «один-против-всех» (обучением классификатора отличать каждый класс от всех других) [10]. В библиотеке Accord.Net для первой стратегии предназначается взятый за основу класс *MulticlassSupportVectorLearning*.

Опишем далее спроектированную архитектуру программного средства (выполняемый файл которого получил название «BinArchId.exe»), поделив ее на два качественно разных слоя: функциональный – отвечающий за ее декомпозицию на модули с определенным функционалом и их взаимосвязи; и информационный – связывающий информационные потоки, циркулирующие в программном средстве и на его границах.

Функциональный слой архитектуры

Исходя из общей схемы работы способа идентификации, был спроектирован следующий состав модулей программного средства и их взаимосвязь в виде функционального слоя архитектуры (рисунок 2).

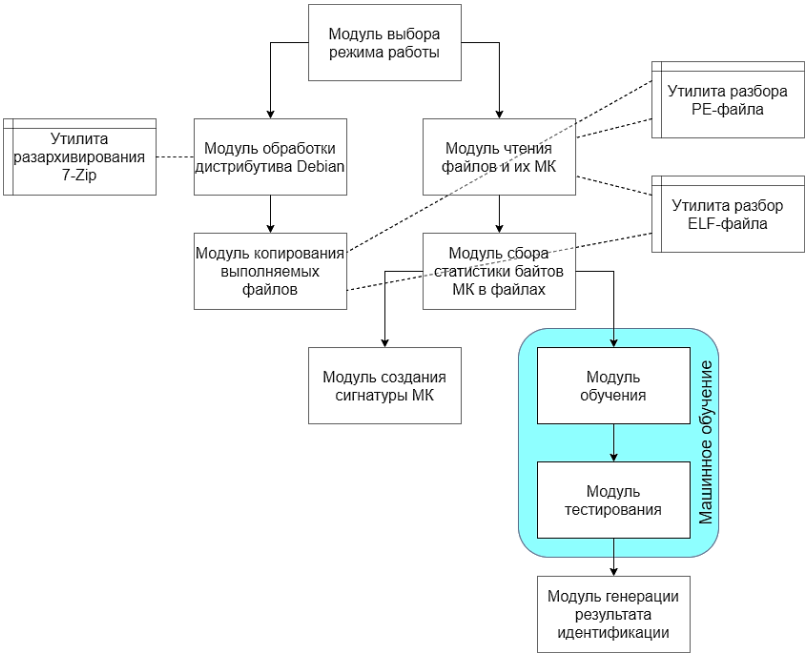

Рис. 2. Функциональный слой архитектуры программного средства

Fig. 2. Functional Layer of Software Architecture

Слой состоит из следующих модулей и утилит (первые выполняют основной функционал способа, а вторые – более общий и вспомогательный функционал, используемый первыми).

«Модуль выбора режима работы» – отвечает за раздельное выполнение этапов способа, что соответствует одному из 3-х режимов работы (о режимах – ниже). С его выполнения начинается работа способа.

«Модуль обработки дистрибутива Debian» – выполняет основную логику распаковки дистрибутива, обрабатывая все каталоги, обнаруживая архивы, распаковывая их (с помощью «Утилиты разархивации 7-Zip»). Модуль выполняется первым на Этапе 1.

«Модуль копирования выполняемых файлов» – вызывается предыдущим модулем, определяет выполняемые файлы по заголовкам (с помощью «Утилиты разбора PE-файла» и «Утилиты разбор ELF-файла») и копирует их в заданную директорию. Модуль выполняется последним на Этапе 1.

«Модуль чтения файлов и их МК» – читает указанные файлы, выделяя в них МК по алгоритму, зависящему от настройки режима: или из кодовых секций, полученных из заголовка файла (PE или ELF); или считая, что файл полностью состоит из МК – режим «X». Модуль выполняется первым на Этапах 2 и 3.

«Модуль сбора статистики байтов МК в файлах» – используя полученные предыдущим модулем секции с МК каждого их файлов, собирает статистику появления в них различных байт и производит их нормировку по имеющему наибольшее значение, получая тем самым частотно-байтовое распределение МК. Модуль общий для Этапов 2 и 3.

«Модуль создания сигнатуры МК» – используя нормированную статистику байт, полученную предыдущем модулем, формирует итоговую частотную сигнатуру МК для файлов исходного дистрибутива Debian. Модуль выполняется последним на Этапе 2.

«Модуль обучения» – производит настройку внутренней модели машинного обучения (отме-

чено на рисунке голубым фоном) по прецедентам: «частотно-байтовое распределение МК» → «архитектура МК», – используя загруженные сигнатуры. Модуль относится к Этапу 2.

«Модуль тестирования» – производит многоклассовую классификацию частотно-байтовых распределений МК тестируемых файлов, полученных предыдущим модулем, с загруженными сигнатурами (для соответствующих архитектур) с помощью методов машинного обучения. Модуль относится к Этапу 3.

«Модуль генерации результата идентификации» – используя результаты классификации от предыдущего модуля, формирует итоговый отчет в виде соотношений «Файл → Архитектура», а также таблицы вероятности отношения каждого тестируемого файла к соответствующей архитектуре. Модуль выполняется последним на Этапе 3.

«Утилита разархивирования 7-Zip» – осуществляет разархивирование файлов, запакованных с помощью архиватора 7-Zip [11] (в таком формате хранятся некоторые файлы из дистрибутива Debian).

«Утилита разбор ELF-файла» – производит проверку наличия в файле заголовка ELF, а также чтение его секций, в том числе с МК.

«Утилита разбор PE-файла» – производит проверку наличия в файле заголовка PE, а также чтение его секций, в том числе с МК.

Информационный слой архитектуры

Преобразование информационных потоков при работе программного средства может быть показано на следующей схеме информационного слоя архитектуры (рисунок 3). Схема достаточно хорошо понятна и не нуждается в пояснениях, поскольку отражает как работу самого способа, так и функциональный слой архитектуры программного средства идентификации; также она соответствует принципам работы методов машинного обучения в части классификации.

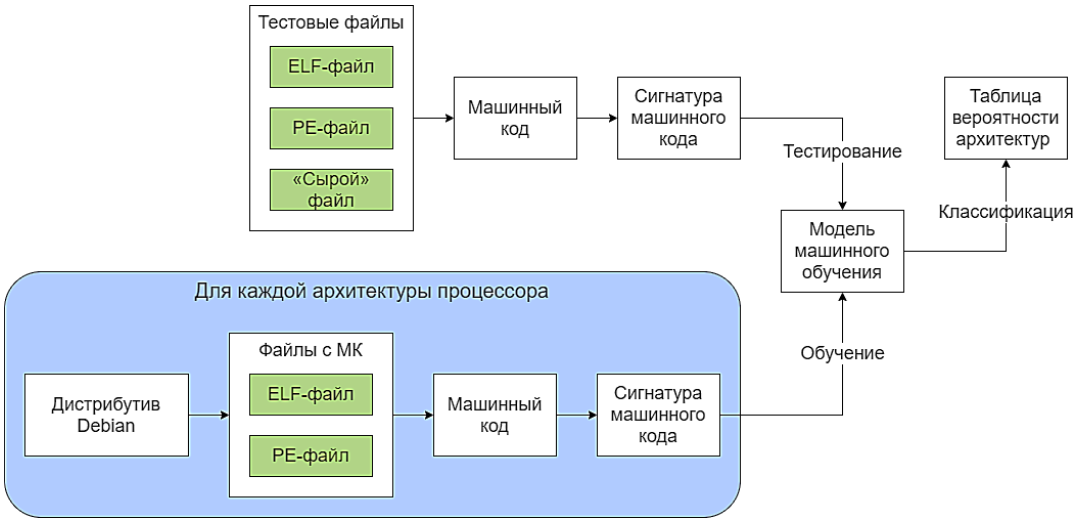

Рис. 3. Информационный слой архитектуры программного средства

Fig. 3. Information Layer of Software Architecture

Режимы работы

Поскольку предполагается, что этапы способа должны выполняться раздельно, то программное средство спроектировано для работы в 3 режимах, каждый из которых соответствует своему этапу. Режимами работы программного средства являются следующие.

Режим 1 – «Распаковка Debian», в результате которого файлы из дистрибутива Debian, содержащие МК, распаковываются в заданную директорию. В качестве входных данных указывается путь к диску, на который монтирован образ дистрибутива. В качестве выходных данных указывается директория для распаковки. Так, после запуска программного средства в данном режиме (аргумент *UnpackMode*) с помощью командной строки:

```
> BinArchId.exe UnpackMode G D:\Debian\i386\
```

дистрибутив Debian на диске «G:\» будет проанализирован на предмет выполняемых файлов, которые будут распакованы в директорию «D:\Debian\i386\» (следуя из названия, для архитектуры процессора *i386*) с сохранением исходной структуры директорий. Остальные (невыполняемые файлы) будут пропущены.

Режим 2 – «Вычисление сигнатур», в результате которого МК всех файлов одной процессорной архитектуры (предварительно распакованных из дистрибутива Debian) будут проанализированы с целью сбора частот появления байт и вычисления сигнатуры класса МК, которая будет сохранена в специальном файле. Так, после запуска программного средства в данном режиме (аргумент *SignatureMode*) с помощью командной строки:

```
> BinArchId.exe SignatureMode D:\Debian\i386\
D:\Debian\i386.sig
```

оно найдет все файлы в директории «D:\Debian\i386\», вычислит их общую сигнатуру и запишет в файл «D:\Debian\i386.sig».

Режим 3 – «Идентификация МК», в результате которого будут загружены все заданные сигнатуры МК с указанием их архитектур, а также файлы с МК из указанной директории. Используя сигнатуры, метод машинного обучения по многоклассовой классификации вычислит вероятность отнесения каждого из МК к одной из переданных архитектур. Так, после запуска программного средства в данном режиме (аргумент *TestMode*) с помощью командной строки:

```
> BinArchId.exe TestMode D:\Test\
amd64=D:\Debian\amd64.sig
i386=D:\Debian\i386.sig
mipsel=D:\Debian\mipsel.sig
```

оно загрузит из директории «D:\Debian\» сигнатуры «amd64.sig», «i386.sig» и «mipsel.sig» для архитектур 64-битного Intel/AMD, 32-битного Intel и 64-битного MIPS с порядком байтов от младшего к старшему, затем получит МК файлов из директории «D:\Test\», и сгенерирует отчет об идентификации в виде списка и расширенной таблицы со следующей интерпретацией: строки = идентифицируемые файлы, столбцы = архитектуры загруженных сигнатур, ячейки = вероятности отнесения файла-строки к архитектуре-столбцу. Для запуска вариации режима, при котором файлы будут интерпретироваться, как полностью состоящие из МК, необходимо в качестве первого аргумента передать *TestModeX*.

3. Тестирование программного средства идентификации

Сценарий 1. Распаковка Debian

Запустим программное средство в режиме «Распаковки Debian» (*UnpackMode*), передав на вход путь к примонтированному диску с дистрибутивом Debian и указав директорию для распаковки выполняемых файлов. Для примера, в качестве диска с дистрибутивом возьмем «G:\», а в качестве директории – «D:\Debian\i386». Начало и конец листинга, выведенного программным средством при работе в режиме, будет следующим (здесь и далее для удобства листинг откорректирован путем удаления отладочной информации и замены средних записей на многоточие):

```
  Unzip: 'G:\pool\main\a\acl\libacl1_2.2.53-
4_i386.deb' ->
D:\Debian\i386\pool\main\a\acl\libacl1_2.2.53-
4_i386.deb
  Unzip:
'D:\Debian\i386\pool\main\a\acl\libacl1_2.2.53-
4_i386.deb\data.tar' ->
D:\Debian\i386\pool\main\a\acl\libacl1_2.2.53-
4_i386.deb\data.tar_
  ...
  Unzip:
'G:\pool\main\z\zlib\zlib1g_1.2.11.dfsg-
1_i386.deb' ->
D:\Debian\i386\pool\main\z\zlib\zlib1g_1.2.11.d
fsg-1_i386.deb
  Unzip:
'D:\Debian\i386\pool\main\z\zlib\zlib1g_1.2.11.
dfsg-1_i386.deb\data.tar' ->
D:\Debian\i386\pool\main\z\zlib\zlib1g_1.2.11.d
fsg-1_i386.deb\data.tar_
```

В логе указаны архивы, вложенные друг в друга, и пути их распаковки.

Сценарий 2. Вычисление сигнатуры

Запустим программное средство в режиме «Вычисление сигнатуры» (*SignatureMode*), передав на вход путь к директории с распакованными выполняемыми файлами одной архитектуры, а также путь к файлу для сохранения сигнатуры. Для примера в качестве архитектуры дистрибутива возьмем архитектуру *i386* (файлы которой ранее распакованы в директорию «D:\Debian\i386\»). Результатом работы программного средства будет следующий листинг:

```
  Elf:
D:\Debian\i386\pool\main\a\acl\libacl1_2.2.53-
4_i386.deb\data.tar\usr\lib\i386-linux-
gnu\libacl.so.1.1.2253
  Elf:
D:\Debian\i386\pool\main\a\acpi\acpi_1.7-
1.1_i386.deb\data.tar\usr\bin\acpi
  ...
  Elf: D:\Debian\i386\pool\main\x\xz-utils\xz-
utils_5.2.4-1_i386.deb\data.tar\usr\bin\xz
  Elf:
D:\Debian\i386\pool\main\z\zlib\zlib1g_1.2.11.d
fsg-1_i386.deb\data.tar\lib\i386-linux-
gnu\libz.so.1.2.11
```

В логе каждая строка начинается с типа заголовка выполняемого файла (ELF или PE), за которым идет полный путь к файлу. Пример сигнатуры для процессорной архитектуры *i386* по всем выполняемым файлам дистрибутива Debian следующий:

```
0    1      35109689
1    0.1143608250132 89     4015173
2    0.0480500695975974     1687023
...
253  0.0223295626457984     783984
254  0.0394533258326498     1385194
255  0.600960834486458      21099548
```

где первое число в строке означает значение байта, второе – долю значения байта по сравнению с другими значениями, а третье – общее число байт с данным значением. Как хорошо видно, в файлах больше всего содержится байт со значением 0 (первая строка, для которой доля максимальна и равна 1).

Сценарий 3. Идентификация архитектуры

Запустим программное средство в режиме «Идентификация архитектуры» (*TestMode* или

TestModeX), передав на вход путь к директории с тестируемыми файлами, а также все процессоры архитектур и их сигнатуры. Для примера и наглядности работы в качестве файлов возьмем Топ-10 рассматриваемых архитектур (в виде соответствующих им сигнатур, полученных ранее) и такое же количество идентифицируемых файлов, поименованных в форме *test.{arch}*, где *{arch}* – название архитектуры. Результатом работы программного средства будет следующий листинг:

```
  test.amd64 -> (amd64)
  test.arm64 -> (arm64)
  ...
  test.ppc64el -> (ppc64el)
  test.s390x -> (s390x)

  test.amd64 -> amd64=0.71 arm64=0.03 ar-
mel=0.04 armhf=0.04 i386=0.05 mips=0.02
mips64el=0.03 mipsel=0.02 ppc64el=0.02
s390x=0.03
  test.arm64 -> amd64=0.13 arm64=0.53 ar-
mel=0.05 armhf=0.05 i386=0.05 mips=0.04
mips64el=0.04 mipsel=0.04 ppc64el=0.04
s390x=0.04
  ...
  test.ppc64el -> amd64=0.05 arm64=0.06 ar-
mel=0.06 armhf=0.06 i386=0.06 mips=0.06
mips64el=0.06 mipsel=0.06 ppc64el=0.41
s390x=0.14
  test.s390x -> amd64=0.06 arm64=0.06 ar-
mel=0.05 armhf=0.05 i386=0.06 mips=0.06
mips64el=0.06 mipsel=0.06 ppc64el=0.06
s390x=0.48
```

Первая часть листинга содержит результаты идентификации в виде имени файла и соответствующей ему архитектуры, а вторая часть приводит вероятности отнесения каждого файла к архитектуре (очевидно, что, в первом листинге приводятся архитектуры с наибольшей из второго вероятностью). Так, в листинге отображена информация для файлов *test.[amd64|arm64|ppc64el|s390x]*. Необходимо уточнить, что хотя тестируемые файлы и содержат заголовки с типом архитектуры, однако эта информация не используется программным средством; анализу подвергаются только секции с кодом.

Результаты идентификации, сведенные вместе, представлены в таблице 1.

ТАБЛИЦА 1. Вероятности отнесения тестовых файлов к архитектурам процессоров

TABLE 1. Probabilities of Assigning Test Files to Processor Architectures

Тестовые файлы	Архитектуры процессора									
	amd64	*arm64*	*armel*	*armhf*	*i386*	*mips*	*mips64el*	*mipsel*	*ppc64el*	*s390x*
test.amd64	0,71	0,03	0,04	0,04	0,05	0,02	0,03	0,02	0,02	0,03
test.arm64	0,13	0,53	0,05	0,05	0,05	0,04	0,04	0,04	0,04	0,04
test.armel	0,09	0,11	0,25	0,07	0,07	0,08	0,08	0,08	0,09	0,08
test.armhf	0,02	0,01	0,00	0,87	0,01	0,02	0,02	0,02	0,02	0,02
test.i386	0,03	0,04	0,04	0,04	0,58	0,05	0,06	0,05	0,05	0,07
test.mips	0,09	0,07	0,06	0,05	0,05	0,41	0,03	0,10	0,03	0,10
test.mips64el	0,11	0,08	0,06	0,06	0,06	0,11	0,27	0,06	0,07	0,12
test.mipsel	0,09	0,07	0,06	0,05	0,05	0,41	0,03	0,10	0,03	0,10
test.ppc64el	0,05	0,06	0,06	0,06	0,06	0,06	0,06	0,06	0,41	0,14
test.s390x	0,06	0,06	0,05	0,05	0,06	0,06	0,06	0,06	0,06	0,48

Примечание. Красным фоном в таблице помечены наибольшие вероятности, соответствующие идентифицированным архитектурам процессора, а зеленым – ближайшая альтернатива.

Как хорошо видно по отчету, идентификация показала хорошие результаты за исключением архитектур mips и mipsel, соответствующих 32 битному MIPS с различным порядком байт. Впрочем, такая ситуация была предсказана ранее при анализе гистограмм частотных сигнатур классов МК [4]. Также следует отметить, что вероятность отнесения к ближайшим альтернативам меньше в 2–3 раза, что можно считать хорошей точностью идентификации. Еще раз повторим, что более точные и формальные оценки будут получены в следующей (заключительной) части цикла.

Заключение

Таким образом, во второй части цикла описана созданная схема способа идентификации архитектуры процессоров МК и сформированы требования к соответствующему программному средству, приведена его архитектура с позиции функциональных модулей и информационных потоков данных. Работа программного средства в трех режимах позволяет разделять выполнение этапов способа, что достаточно значимо с практической точки зрения. Базовое тестирование программного средства для файлов 10 архитектур, рассмотренных в первой части статьи, обосновывает его работоспособность, даже несмотря на неразличимость близких архитектур mips и mipsel. Также можно утверждать, что программное средство удовлетворяет поставленным требованиям: результативность обоснована рабочим примером по идентификации, оперативность – режимами рабо-

ты программного средства и применением многоклассового классификатора на базе SVM (по стратегии «один-против-одного»), а ресурсоэкономность – полной автоматизацией работы и простотой создания обучающей выборки.

Для получения количественных оценок работоспособности программного средства необходимо провести его полноценное тестирование и вычисление показателей таких характеристик качества, как точность, полнота, аккуратность, ошибка и F-мера [12]. Также интересным, с научной и практической точки зрения, будет исследование границ применимости способа, например, для файлов без заголовков [13–14], малого размера, разрушенных [15], модифицированных [16], а также содержащих МК нескольких архитектур (как, например, в магистральных маршрутизаторах). Этому будет посвящена третья, заключительная, часть цикла статей.

Окончание следует ...

Список используемых источников

1. Буйневич М.В., Васильева И.Н., Воробьев Т.М., Гниденко И.Г., Егорова И.В. и др. Защита информации в компьютерных системах: монография. СПб.: Санкт-Петербургский государственный экономический университет, 2017. 163 с.
2. Buinevich M., Izrailov K., Vladyko A. Metric of vulnerability at the base of the life cycle of software representations // Proceedings of the 20th International Conference on Advanced Communication Technology (ICACT, Chuncheon-si Gangwon-do, South Korea, 11–14 February 2018). IEEE, 2018. PP. 1–8. DOI:10.23919/ICACT.2018.8323940
3. Безмалый В. Антивирусные сканеры // Windows IT Pro/ RE. 2014. № 4. С. 52.
4. Буйневич М.В., Израилов К.Е. Идентификация архитектуры процессора выполняемого кода на базе машинного обучения. Часть 1. Частотно-байтовая модель // Труды учебных заведений связи. 2020. Т. 6. № 1. С. 77–85. DOI:10.31854/1813-324X-2020-6-1-77-85
5. Спиридонов В. Задачи, эвристики, инсайт и другие непонятные вещи // Логос. 2014. № 1(97). С. 97–108.
6. Буйневич М.В., Израилов К. Е. Обобщенная модель статического анализа программного кода на базе машинного обучения применительно к задаче поиска уязвимостей // Информатизация и Связь. 2020. №. 2. С. 143–152. DOI:10.34219/2078-8320-2020-11-2-143-152
7. Файлы образов Debian версии 10.3.0 // Debian. URL: https://www.debian.org/distrib/netinst.ru.html (дата обращения: 26.06.2020)
8. Федоров Д.Ю. Программирование на языке высокого уровня Python: учебное пособие. Москва: Издательство Юрайт, 2019. Сер. 60. Бакалавр. Прикладной курс. 161 с.
9. Пижевский М.К. Инструменты машинного обучения // Modern Science. 2020. № 1–1. С. 435–438.
10. Браницкий А.А., Саенко И.Б. Методика многоаспектной оценки и категоризации вредоносных информационных объектов в сети Интернет // Труды учебных заведений связи. 2019. Т. 5. № 3. С. 58–65. DOI:10.31854/1813-324X-2019-5-3-58-65
11. Файловый архиватор 7-Zip // 7-Zip. URL: https://www.7-zip.org/ (дата обращения: 26.06.2020)
12. Шелухин О.И., Симонян А.Г., Ванюшина А.В. Влияние структуры обучающей выборки на эффективность классификации приложений трафика методами машинного обучения // T-Comm: Телекоммуникации и транспорт. 2017. Т. 11. № 2. С. 25–31.
13. Трофименков А.К., Трофименков С.А., Пимонов Р.В. Алгоритмизация обработки файлов для их идентификации при нарушении целостности данных // Системы управления и информационные технологии. 2020. № 2(80). С. 82–85.
14. Антонов А.Е., Федулов А.С. Идентификация типа файла на основе структурного анализа // Прикладная информатика. 2013. № 2(44). С. 068–077.
15. Касперски К. Как спасти данные, если отказал жесткий диск // Системный администратор. 2005. № 9(34). С. 80–87.
16. Штеренберг С.И., Андрианов В.И. Варианты модификации структуры исполнимых файлов формата PE // Перспективы развития информационных технологий. 2013. № 16. С. 134–143.

* * *

Identification of Processor's Architecture of Executable Code Based on Machine Learning. Part 2. Identification Method

M. Buinevich[1, 2], **K. Izrailov**[1, 3]

[1]The Bonch-Bruevich Saint-Petersburg State University of Telecommunications, St. Petersburg, 193232, Russian Federation
[2]Saint-Petersburg University of State Fire Service of Emercom of Russia, St. Petersburg, 195105, Russian Federation
[3]St. Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences, St. Petersburg, 199178, Russian Federation

Article info
DOI:10.31854/1813-324X-2020-6-1-104-112
Received 1st June 2020
Accepted 24th June 2020

For citation: Buinevich M., Izrailov K. Identification of Processor's Architecture of Executable Code Based on Machine Learning. Part 2. Identification method. *Proc. of Telecom. Universities.* 2020;6(2):104–112. (in Russ.) DOI:10.31854/1813-324X-2020-6-2-104-112

Abstract: *This article shows us the study results of a method for identifying the processor architecture of an executable code based on machine learning. In the second part of the series of articles, a three-stage scheme of the method and the corresponding software are synthesized. The functional and information layer of the architecture of the tool, as well as its operation modes, are described. Basic testing of the tool is carried out and the results of its work are given. By the example of identification of files with machine code of various architectures, the efficiency of the proposed method and means is substantiated.*

Keywords: *information security, machine code, processor architecture, machine learning, frequency-byte model, code signature, processor identification method, software*

References

1. Buynevich M.V., Vasilieva I.N., Vorobyov T.M., Gnidenko I.G., Egorova I.V. et al. *Information Security in Computer Systems*. St. Petersburg: Saint Petersburg Electrotechnical University "LETI" Publ.; 2017. 163 p. (in Russ.)
2. Buinevich M., Izrailov K., Vladyko A. Metric of vulnerability at the base of the life cycle of software representations. *Proceedings of the 20th International Conference on Advanced Communication Technology, ICACT, 11–14 February 2018, Chuncheon-si Gangwon-do, South Korea*. IEEE; 2018. p.1–8. DOI:10.23919/ICACT.2018.8323940
3. Bezmalyy V. Antivirus scanners. *Windows IT Pro/ RE*. 2014;4:52. (in Russ.)
4. Buinevich M., Izrailov K. Identification of Processor's Architecture of Executable Code Based on Machine Learning. Part 1. Frequency Byte Model. *Proc. of Telecom. Universities*. 2020;6(1):77-85. (in Russ.) DOI:10.31854/1813-324X-2020-6-1-77-85
5. Spiridonov V. Problems, Heuristics, Insight and Other Strange Things. *Logos*. 2014;1(97):97–108. (in Russ.)
6. Buinevich M., Izrailov K. A Generalized Model of Static Analysis of Program Code Based on Machine Learning for the Vulnerability Search Problem. *Informatizatsiya i Svyaz'*. 2020;2:143–152 (in Russ.) DOI:10.34219/2078-8320-2020-11-2-143-152
7. *Debian.* Debian Image Files Version 10.3.0. Available from: https://www.debian.org/distrib/netinst.ru.html (in Russ.) [Accessed 26th June 2020]
8. Fedorov D.Yu. *Python High-Level Programming*.Moscow: Yurayt Publ.; 2019. 161 p. (in Russ.)
9. Pizhevsky M.K. Machine Learning Tools. *Modern Science*. 2020;1–1:435–438. (in Russ.)
10. Branitskiy A., Saenko I. The Technique of Multi-Aspect Evaluation and Categorization of Malicious Information Objects on the Internet. *Proc. of Telecom. Universities*. 2019;5(3):58–65. (in Russ.) DOI:10.31854/1813-324X-2019-5-3-58-65
11. *7-Zip*. File Archiver 7-Zip. Available from: https://www.7-zip.org/ [Accessed 26th June 2020]
12. Sheluhin O.I., Simonyan A.G., Vanyushina A.V. Influence of training sample structure on traffic application efficiency classification using machine-learning methods. *T-Comm.* 2017;11(2):25–31. (in Russ.)
13. Trofimenkov A.K., Trofimenkov S.A., Pimonov R.V. Algorithmization of File Processing for their Identification in Case of Violation of Data Integrity. *Sistemy upravleniya i informatsionnyye tekhnologii*. 2020;2(80):82–85. (in Russ.)
14. Antonov A., Fedulov A. File type identification based on structural analyses. *Journal of Applied Informatics*. 2013;2(44):068–077. (in Russ.)
15. Kaspersky K. How to Save Data if the Hard Drive Fails. *Sistemnyy administrator*. 2005;9(34):80–87.
16. Shterenberg S.I., Andrianov V.I. Options for Modifying the Structure of PE Executable Files. *Perspektivy razvitiya informatsionnykh tekhnologiy*. 2013;16:134–143. (in Russ.)

Сведения об авторах:

БУЙНЕВИЧ Михаил Викторович — доктор технических наук, профессор, профессор кафедры безопасности информационных систем Санкт-Петербургского государственного университета телекоммуникаций им. проф. М.А. Бонч-Бруевича, профессор кафедры прикладной математики и информационных технологий Санкт-Петербургского университета государственной противопожарной службы МЧС России, bmv1958@yandex.ru
https://orcid.org/0000-0001-8146-0022

ИЗРАИЛОВ Константин Евгеньевич — кандидат технических наук, доцент кафедры защищенных систем связи Санкт-Петербургского государственного университета телекоммуникаций им. проф. М.А. Бонч-Бруевича, старший научный сотрудник Санкт-Петербургского института информатики и автоматизации Российской академии наук, konstantin.izrailov@mail.ru
https://orcid.org/0000-0002-9412-5693