УДК 004.056.53

© 2018 г. **Н.И. Сельвесюк**, д-р техн. наук,
А.С. Островский, канд. техн. наук
(Московский государственный технический университет им. Н.Э. Баумана),
П.В. Русанов,
М.О. Комахин
(ООО «ИНФОРИОН»)

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ СИСТЕМЫ ЭМУЛЯЦИИ ПРОГРАММНОГО ИНТЕРФЕЙСА ПРИЛОЖЕНИЯ В ЗАДАЧАХ ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННОЙ БЕЗОПАСНОСТИ*

В статье рассматривается объектно-ориентированный подход к созданию системы для программной эмуляции вызовов сторонних функций из пользовательских приложений, разработанных под произвольные операционные системы и скомпилированные под различные процессорные архитектуры. Приведено краткое описание основных принципов построения программных интерфейсов. Подробно рассмотрены подходы к эмуляции программного интерфейса приложения.

Ключевые слова: объектно-ориентированное проектирование, информационная безопасность, анализ программного обеспечения.

DOI: 10.22250/isu.2018.56.3-13

Введение

При решении ряда задач, связанных с обеспечением информационной безопасности автоматизированных систем, возникает необходимость анализа программного обеспечения этих систем на предмет наличия недокументированной функциональности и возможных ошибок. Рост функциональности и сложности программного обеспечения приводит к тому, что статический анализ двоичного кода приложения становится все более трудозатратным и часто неэффективным.

* Работа выполнена при финансовой поддержке РФФИ, гранты № 16-08-00311-а, № 18-08-00486-а.

Преимущественным вариантом может быть сочетание статического и динамического анализов. Для общедоступных платформ и операционных систем динамический анализ исполняемого кода может выполняться непосредственно на вычислительной системе исследователя. При существовании вероятности разрушающих программных воздействий в исследуемом приложении уместно использовать современные средства виртуализации для выполнения динамического анализа. Однако даже в случае применения виртуальной среды препятствием для запуска приложения может стать его ориентированность на специальное аппаратное и/или программное обеспечение (например, на операционную систему специального назначения). При этом в случае отсутствия сведений о целевой вычислительной системе для автоматизации определения ее архитектуры может быть применен подход, представленный в [1].

С учетом упомянутых ограничений более безопасным и продуктивным способом динамического анализа приложений может стать система низкоуровневой эмуляции [2]. Однако в случае использования специализированной операционной системы с большой кодовой базой полная низкоуровневая эмуляция всей системы будет неэффективна, так как создаст повышенную нагрузку на работу процессора и использование оперативной памяти. Поэтому при исследовании отдельного приложения или группы приложений предлагается использовать эмуляцию программного окружения (библиотек, драйверов).

В качестве основы для создания системы эмуляции использован низкоуровневый программный эмулятор встраиваемых систем, имеющий следующие важные особенности: возможность полного управления состоянием системы и портов ввода-вывода, возможность создания виртуальной среды из нескольких эмулируемых систем, наличие GDB-интерфейса [3] для обеспечения интеграции со сторонними исследовательскими инструментами. Для реализации системы эмуляции выбран язык программирования Kotlin, основанный на виртуальной машине Java (JVM) [4]. Выбор данного языка обусловлен рядом преимуществ: скорость и удобство разработки, кроссплатформенность и поддержка библиотек языка Java.

Основные задачи программных интерфейсов

Программный интерфейс приложения (application programming interface, API) определяет функциональность, которую предоставляет программа (модуль, библиотека), при этом API абстрагирует конечного пользователя от знания о непосредственной реализации данной функциональности [5].

В общем случае API определяет протокол «состыковки» отдельных компонентов как внутри операционной системы и при построении пользовательских

приложений, так и при взаимодействии отдельных систем.

При разработке программного обеспечения обычно используются стандартизированные протоколы взаимодействия, – например, POSIX-API [6], API баз данных, API сокетов Беркли, API графических подсистем, WinAPI и др. При этом могут выстраиваться иерархии компонентов: от API пользовательских приложений до API ядра операционной системы. Аналогичным образом выстраивается стек протоколов TCP/IP, когда каждый следующий уровень стека использует функциональность предыдущего при передаче данных. Обобщенная схема взаимодействия компонентов через API представлена на рис. 1.

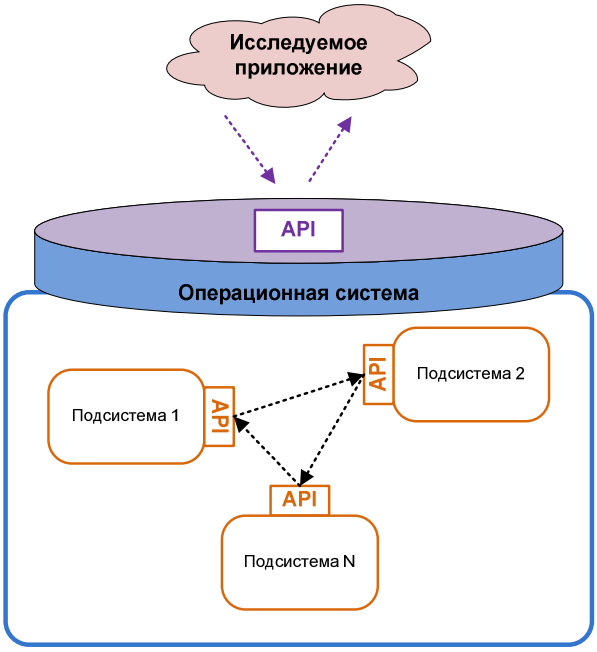

Рис. 1. Взаимодействие компонентов системы посредством API.

При рассмотрении API библиотек функций и классов следует уделить внимание описанию сигнатур и семантик функций.

Сигнатура функции представляет собой часть общего объявления функции, которая позволяет транслятору идентифицировать функцию среди других. Состав сигнатуры может по-разному трактоваться в зависимости от используемого языка программирования, что тесно связано с наличием возможности перегрузки функций в этом языке. Так, простая функция в языке программирования C++ явно идентифицируется компилятором по имени и последовательности аргументов, что является сигнатурой данной функции. Для метода класса в сигнатуру добавляется имя класса.

Семантика функции представляет собой описание того, что данная функция делает. Семантика определяет результат работы функции и условия получения результата. При этом результат работы может зависеть не только от значений аргументов функции, но и от текущего состояния информационной системы. Изменение ее состояния также может быть результатом работы функции. Эти взаимо-

связи также относятся к семантике функции. Полным представлением семантики функции является ее код или математическое определение.

При построении системы эмуляции весь спектр используемых приложением API-функций можно разделить на следующие группы:

критичные функции, обеспечивающие функционирование приложения (работа с файлами, сетью, памятью) – их функциональность должна быть эмулируема в большей степени;

функции, влияющие на внутреннее состояние приложения (инициализация внутренних структур данных, диагностика периферийных устройств) – их функциональность может быть эмулирована лишь частично;

некритичные функции, не влияющие на внутренние состояния, которые могут быть подменены «функциями-заглушками» (работа с потоками ввода-вывода, журналирование событий).

Наряду с API, дополнительным объектом эмуляции программного окружения должен являться двоичный (бинарный) интерфейс приложений (application binary interface, ABI), представляющий набор соглашений для доступа приложения к операционной системе и другим компонентам. Основная задача ABI – осуществлять переносимость исполняемого кода между машинами с одинаковыми ABI. Если API регламентирует совместимость на уровне исходного кода, то ABI позволяет компоновщику объединять откомпилированные модули без перекомпиляции всего кода и определяет двоичный интерфейс различных компонентов.

Разделение уровней API, ABI и архитектуры набора команд (instruction set architecture, ISA) схематично представлено на рис. 2.

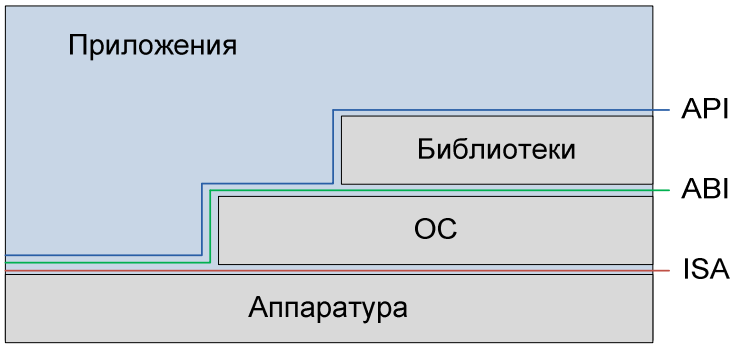

Рис. 2. Уровни и интерфейсы между ними.

Подробное описание реализации эмулятора API-функций приведено на примере эмулятора операционной системы специального назначения, которая устанавливается на встраиваемые системы.

Общая архитектура виртуальной системы

Цель создания виртуальной эмулируемой операционной системы (virtual emulation operating system, VEOS) и виртуального программного интерфейса

(virtual application programming interface, VAPI) состоит в том, чтобы бинарный код исследуемого приложения полностью выполнялся низкоуровневым эмулятором, а функции операционной системы и отдельных библиотек эмулировались, тем самым ускоряя работу приложения по сравнению с полной низкоуровневой эмуляцией ОС.

Технологии VEOS и VAPI позволяют выполнять динамический анализ ПО, которое может выполняться под управлением как универсальных, так и специализированных операционных систем и библиотечных модулей.

Структурная схема уровней VEOS приведена на рис. 3.

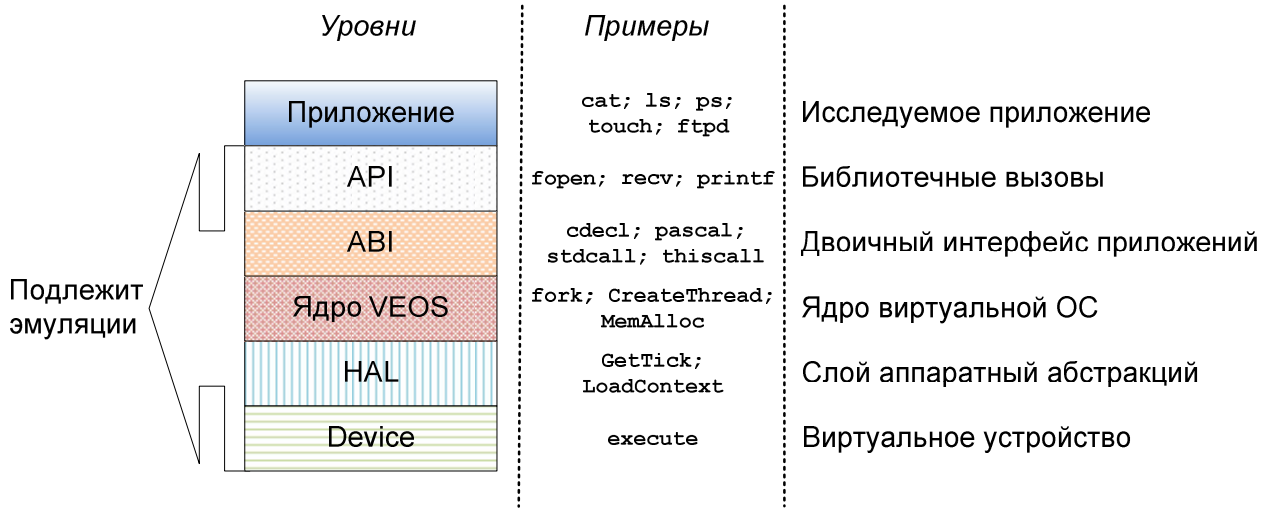

Рис. 3. Структурная схема уровней VEOS.

Уровень API предоставляет приложению доступ к системным вызовам, через которые происходит взаимодействие с операционной системой. Уровень ABI (Application Binary Interface) определяет набор соглашений по обмену данными между операционной системой и приложением – способ и последовательность передачи аргументов и возвращаемого значения и др. Уровень VEOS осуществляет работу по управлению процессами, осуществлению перехвата системных вызовов от приложений и выполнению функций, эмулирующих системное API. Вся работа с памятью эмулятора, регистрами и прочими объектами устройства происходит на уровне ADevice [2]. Вся аппаратно зависимая функциональность, такая как создание, сохранение и загрузка контекста процесса, получение аргументов функции и прочее, вынесена в объект класса, унаследованный от абстрактного класса HAL (hardware abstraction layer) – слой аппаратных абстракций. Этот уровень зависит от процессорной архитектуры и дает возможность абстрагировать эмуляцию операционной системы от аппаратного уровня, позволяя использовать VEOS с разными процессорными архитектурами. Объектно-ориентированный подход позволяет абстрагировать VEOS от аппаратной платформы, используя наследников класса HAL.

Общая архитектура API-слоя

При реализации VAPI выделено несколько частей, выполняющих свои специфические задачи:

HAL (hardware abstraction layer) – позволяет абстрагироваться от специфики конкретной процессорной архитектуры (набор регистров, аппаратное обеспечение);

ABI-слой – позволяет абстрагировать реализацию API-функции от конкретной ОС и аппаратной архитектуры;

API файловой системы – эмуляция работы файловой системы;

функции сетевого взаимодействия – эмуляция сетевого взаимодействия;

функции, специфичные для конкретной ОС и библиотеки – здесь могут реализовываться как POSIX-функции (их назначение известно), так и функции, специфичные для конкретной ОС или библиотеки (их назначение не всегда известно, для эмуляции требуется дополнительное исследование поведения данных функций в конкретной системе).

Каждую из этих частей удобно реализовать в виде отдельного класса или группы классов.

Диаграмма классов API, используемых VEOS, представлена на рис. 4.

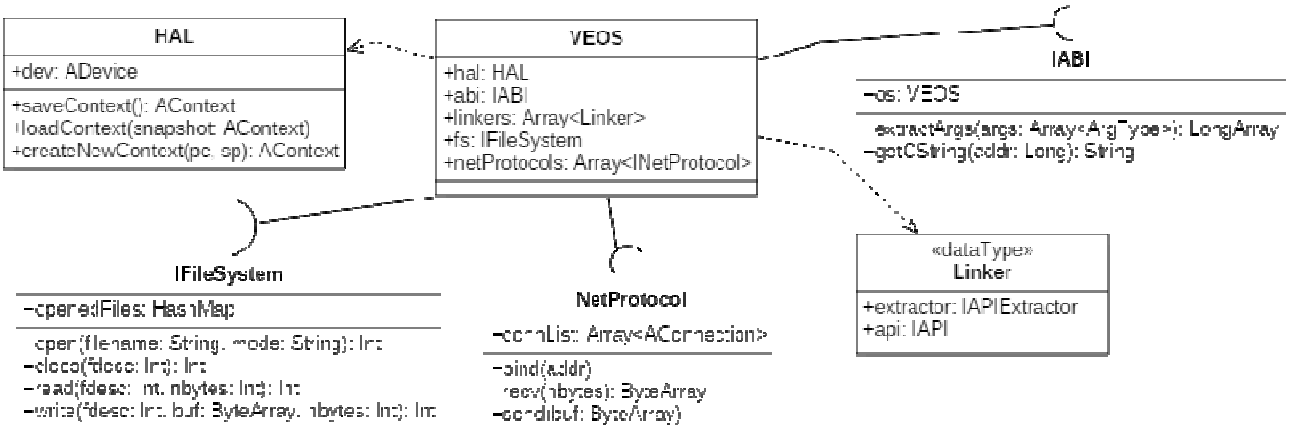

Рис. 4. Диаграмма классов API VEOS.

Доступ к периферийным устройствам

Важным свойством предлагаемого подхода к эмуляции является максимальное абстрагирование приложения, API-функций и ядра VEOS от аппаратной составляющей. Операционная система должна предоставлять унифицированный аппаратно-независимый интерфейс доступа к периферийным устройствам, используя атрибут класса HAL. При этом базовым требованием операционной системы к объекту класса HAL является сохранение и загрузка контекста (набор регистров) при переключении задач.

Диаграмма классов аппаратных абстракций представлена на рис. 5.

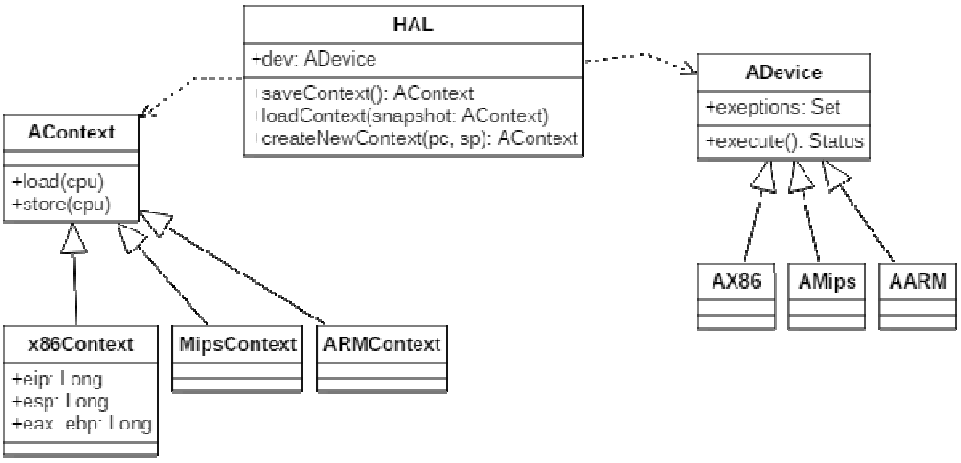

Рис. 5. Диаграмма классов аппаратных абстракций.

Для удобства работы с API-функциями могут быть дополнительно реализованы классы StackStream и RegisterStream, предоставляющие потоковый доступ к стеку и регистрам соответственно.

Поддержка двоичного интерфейса приложений

В общем случае двоичный интерфейс приложений регламентирует: использование регистров процессора; состав и формат системных вызовов и вызовов одного модуля другим; формат передачи аргументов и возвращаемого значения при вызове функции.

Исходя из этого, был разработан интерфейс IABI, основной задачей которого является предоставление методов extractArgs и saveResult. Метод extractArgs служит для получения аргументов контролируемой API-функции операционной системой и передачи их обработчику API-функции. Метод saveResult позволяет операционной системе сохранить результат вызванной функции в зависимости от принятого соглашения. Классы, реализующие интерфейс IABI, дают возможность выполнять одни и те же API-функции на разных платформах, с разными соглашениями о вызовах.

Таким образом, интерфейс IABI является «фильтром» между стандартизированными API и платформозависимыми компонентами.

Диаграмма классов интерфейса IABI представлена на рис. 6.

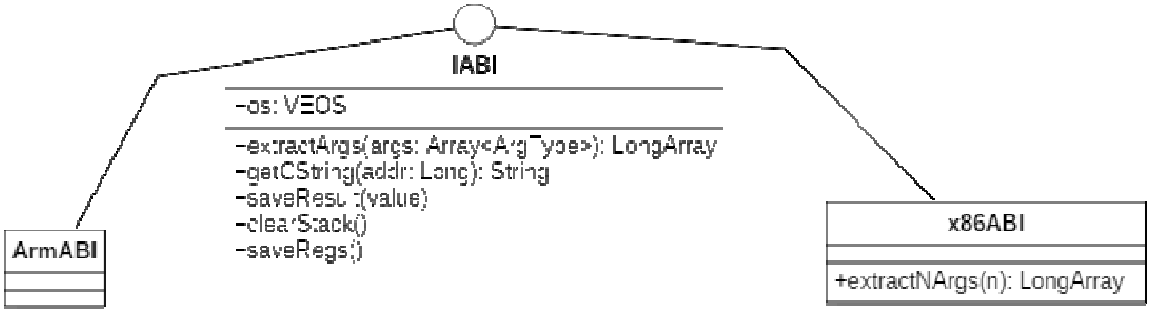

Рис. 6. Диаграмма классов интерфейса IABI.

Получение информации о контролируемых API-функциях

При реализации отдельных классов API-функций необходимо иметь возможность получать информацию об используемых API-функциях исследуемого приложения – их именах и виртуальных адресах.

Поскольку исследуемое приложение может иметь разный конечный бинарный формат (ELF, PE, Mach-O), был разработан универсальный объектный интерфейс IAPIExtractor, который предоставляет список имен функций, требующих эмуляции, и отображение виртуальных адресов функций на эти имена.

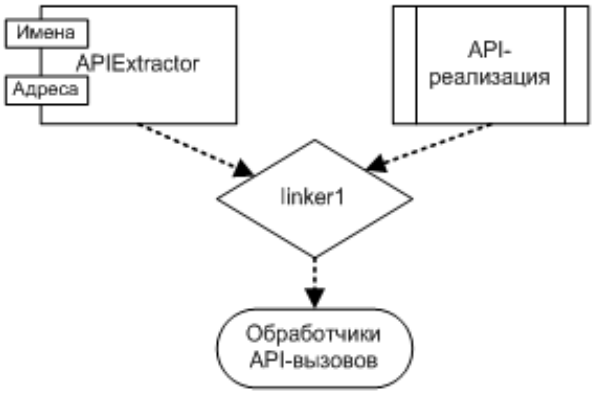

Рис. 7. Общая схема компоновки.

Логическое связывание объектов классов APIExtractor и API реализовано в виде программной структуры Linker (компоновщик). При этом конечная система эмуляции может содержать несколько компоновщиков. Общая схема компоновки представлена на рис. 7.

Для обеспечения универсального интерфейса работы с API-функциями был создан абстрактный класс ASimpleAPI, содержащий «функцию-заглушку» dummyFunc, используемую вместо нереализованных API-функций. Также класс ASimpleAPI предоставляет атрибут apiFuncs – список всех объектов-функций с интерфейсом IFunc. Таким образом, класс ASimpleAPI может служить базой для многих классов API-функций, – например, для класса, реализующего эмуляцию POSIX-функций.

Диаграмма классов интерфейса IAPI представлена на рис. 8.

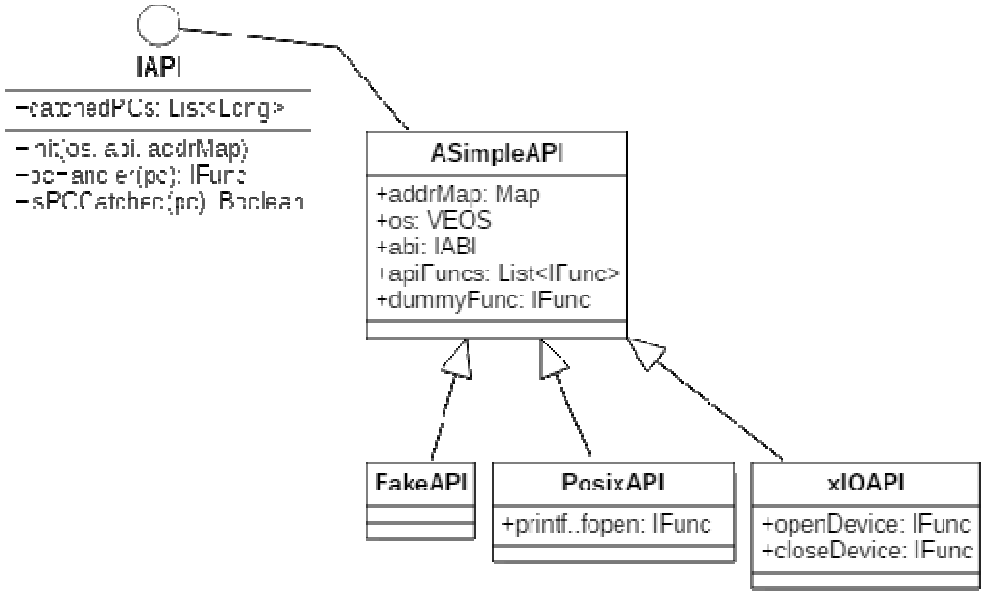

Рис. 8. Диаграмма классов интерфейса IAPI.

Интерфейс IFunc подразумевает предоставление атрибута args и метода exec. Атрибут args представляет собой массив типов данных – фактически, массив длин аргументов функции. Метод exec реализует непосредственную функциональность эмулируемой API-функции и возвращает результат специального класса APIResult. Создание класса APIResult объясняется необходимостью получения дополнительной информации о состоянии системы в ходе выполнения API-функции (блокировка, завершение задачи, системное прерывание), чтобы позволить виртуальной системе обрабатывать это состояние должным образом.

Диаграмма связи API-функций и их метаинформации представлена на рис. 9.

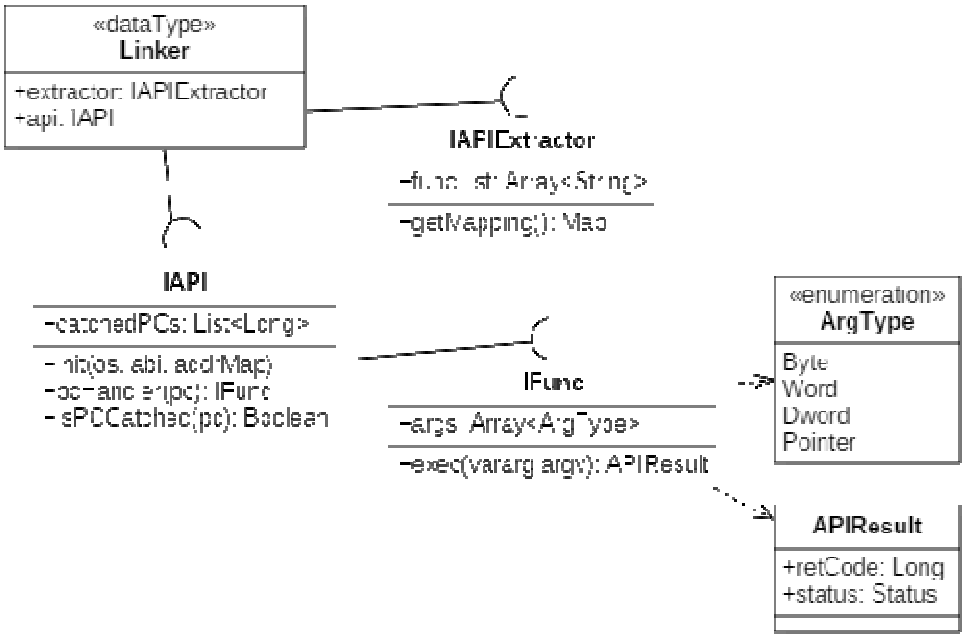

Рис. 9. Диаграмма связей классов IAPI, IAPIExtractor и IFunc.

Поддержка файловых систем

Практически все операционные системы имеют поддержку различных файловых систем. Этот механизм позволяет приложениям иметь стандартный доступ к объектам файловых систем через API операционной системы. Для POSIX-совместимых систем примерами функций для работы с файловой системой могут быть функции fopen, fclose, fgetc, fread, fwrite, fflush и др.

Исходя из этого, был реализован обобщенный интерфейс эмулятора файловой системы IFileSystem, который предоставляет базовые функции открытия/закрытия файлов, чтения/записи массива байтов, абстрагируясь от реализации конкретной файловой системы, которая может проецироваться, например, на реальную файловую систему ОС исследователя или на отдельный файл с образом исследуемой файловой системы.

Диаграмма классов эмулятора файловой системы представлена на рис. 10.

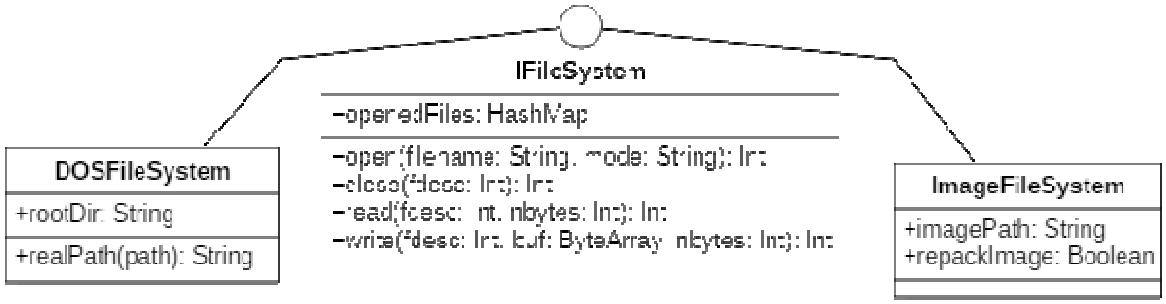

Рис. 10. Диаграмма классов эмулируемой файловой системы.

Поддержка сетевого взаимодействия

По аналогии с другими классами API-функций группа функций для сетевого взаимодействия была обобщена в общий интерфейс INetProtocol, который предоставляет список активных подключений connList, а также функции отправки/получения массива байтов по сети send/recv. Реализация интерфейса INetProtocol может быть представлена, например, классами TcpIp и Udp, которые отражают работу протоколов транспортного уровня TCP и UDP соответственно. При этом каждый класс-протокол с интерфейсом INetProtocol может содержать независимую реализацию физического транспорта данных, – например, «проксирование» данных из виртуальной сети во внешнюю, работа через файл или другой источник данных. Диаграмма классов эмуляции сетевого взаимодействия представлена на рис. 11.

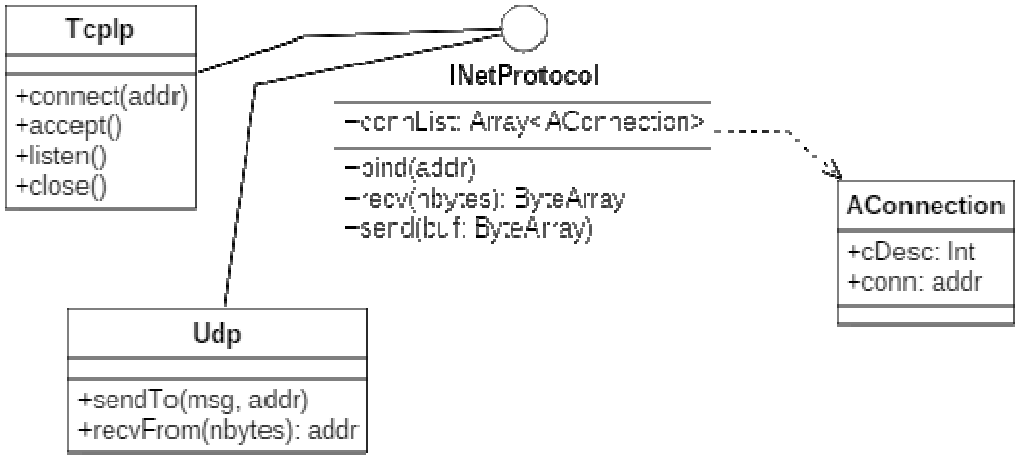

Рис. 11. Диаграмма классов эмуляции сетевого взаимодействия.

Заключение

В статье представлены подходы и результаты разработки программного обеспечения с архитектурой, которая позволяет эмулировать программный интерфейс приложения для произвольной операционной системы с целью динамического исследования пользовательских приложений. Динамическое исследование обеспечивается низкоуровневой программной эмуляцией устройства, ядра

ОС и API, под которые разрабатывалось исследуемое программное обеспечение.

Использование данного программного обеспечения при решении ряда задач, связанных с обеспечением информационной безопасности автоматизированных систем, позволяет повысить достоверность и полноту анализа программного обеспечения этих систем на предмет наличия недокументированной функциональности и возможных ошибок.

ЛИТЕРАТУРА

1. *Сельвесюк Н.И., Островский А.С., Гладких А.А., Аристов Р.С.* Объектно-ориентированное проектирование нейронной сети для автоматизации определения архитектуры вычислительной системы в задачах обеспечения информационной безопасности // Научный вестник НГТУ. – 2016. – С. 133-145.
2. *Давыдов В.Н.* Разработка методик повышения надежности и безопасности встраиваемых вычислительных систем на базе комплексной низкоуровневой программной эмуляции // Наукоемкие технологии и интеллектуальные системы – 2017 / под ред. В.А. Шахнова, А.И. Власова, В.А. Соловьева, В.Г. Перепелицына. – М.: МГТУ, 2017. – С. 56-62.
3. GDB: официальный сайт программного обеспечения. URL: https://www.gnu.org/software/gdb/ (дата обращения: 09.02.2018).
4. Kotlin: официальный сайт языка. URL: https://kotlinlang.org/ (дата обращения: 09.02.2018).
5. *Bierhoff Kevin.* API Protocol Compliance in Object-Oriented Software. CMU Institute for Software Research. URL: https://www.cs.cmu.edu/~kbierhof/thesis/bierhoff-thesis.pdf (дата обращения 09.02.2018).
6. Стандарт POSIX – ISO/IEC 9945 URL: http://www.unix.org/version3 (дата обращения 09.02.2018).

E-mail:
Сельвесюк Николай Иванович – selvesyuk@yandex.ru;
Островский Александр Сергеевич –aleksandr_ostrovsky@mail.ru;
Русанов Павел Васильевич – p.rusanov@inforion.ru;
Комахин Михаил Олегович – m.komakhin@inforion.ru.