DOI: 10.15514/ISPRAS-2022-34(5)-3

Метод восстановления протокольных автоматов по бинарному коду

И.В. Шарков, ORCID: 0000-0002-9767-142X <sharkov@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Реверс-инжиниринг сетевых протоколов широко применяется в задачах анализа безопасности сетевых программ. Задача реинжиниринга протокола состоит из восстановления форматов данных и менее изученного вопроса – восстановления реализованного в программе протокольного автомата, а краеугольным камнем является отсутствие формально понятия состояния протокола. В настоящее время сложились два основных подхода к восстановлению автоматов сетевых протоколов, базирующиеся на использовании различной исходной информации: на основе анализа записанных сетевых трасс и путем анализа бинарного кода программы, реализующего исследуемый протокол. В статье предлагается метод восстановления протокольных автоматов на основе анализа бинарного кода трасс программ в процессе их выполнения. Первой целью работы является описание математической модели протокольного автомата и метода ее проецирования на бинарный код приложения. Вторая цель – описание концепции понятия состояния протокола и правил, указывающих на выполнение переходов, с использованием некоторых «глобальных» объектов трассы программы. Третья преследует способ уточнения протокола с помощью фаззинга в памяти процесса (in-memory fuzzing) с использованием «плавающей» точки запуска порождающего сервера для контроля состояний и управления переходами. Наконец, в статье очерчена схема разработанного набора инструментов и показаны эксперименты по его использованию с реальным VPN-клиентом, подтверждающие состоятельность подхода.

Ключевые слова: сетевой протокол; протокольный автомат; состояние; бинарный код; фаззинг

Для цитирования: Шарков И.В. Метод восстановления протокольных автоматов по бинарному коду. Труды ИСП РАН, том 34, вып. 5, 2022 г., стр. 43-62. DOI: 10.15514/ISPRAS-2022-34(5)-3

Protocol automata recovery method using binary code

I.V. Sharkov, ORCID: 0000-0002-9767-142X <sharkov@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. Security analysis of network programs includes set of reverse engineering tasks of network protocols. Data formats restoring and implemented protocol automaton are the previous task issues. Unlike quite researched problem of formats restoring where there are lots of scientist's papers, finding out the protocol's automaton program implementation looks like terra incognita and the cornerstone is a protocol state description currently undefined. There are two general ways to retrieve the implemented protocol automaton: an analysis of the network traces and looking into binary trace of the target application. This article offers a second one method. The first aim of the paper is the way to describe a mathematical model of a protocol automaton and a method for projecting it onto an executing application binary code. The second is concept of the protocol state definition and a principle to detect the states transitions based on some "global" binary trace objects, are described. Thirdly, there is suggested a protocol automaton precising manner by in-memory fuzzing based on a "floating" fork-server to manage states transitions. Finally, developed toolset's scheme and experiments on its using with a real VPN client, are shown.

Keywords: network protocol; protocol automata; state; binary code; fuzzing
For citation: Sharkov I.V. Protocol automata recovery method using binary code. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 5, 2022. pp. 43-62 (in Russian). DOI: 10.15514/ISPRAS-2022-34(5)-3

1. Введение

Реверс-инжиниринг сетевых протоколов применяется в задачах анализа безопасности сетевых программ. Решения данной задачи позволяет обнаруживать отклонения исследуемого протокола от эталона, выявлять отличия, возникшие в ходе реализации различных программных средств, сравнивать версионные изменения, оценивать совместимость программ, обнаруживать проблемные места.

Восстановление спецификаций недокументированных протоколов и протокольных автоматов путем анализа исходных текстов программ задача весьма трудоемкая, требующая значительного времени [1-3], кроме того, исходные тексты не всегда доступны. Это подталкивает исследователей выбирать другие способы получения информации, позволяющей восстанавливать спецификации протоколов и протокольные автоматы, и создавать автоматизированные средства для решения проблемы.

Пути решения данной задачи в условиях отсутствия доступа к исходным текстам (рис. 1) можно разделить на два направления: анализ сетевых трасс и исследование бинарного кода приложений. При этом следует отметить, что наличие сетевых трасс в ходе анализа бинарного кода играет вспомогательную роль.

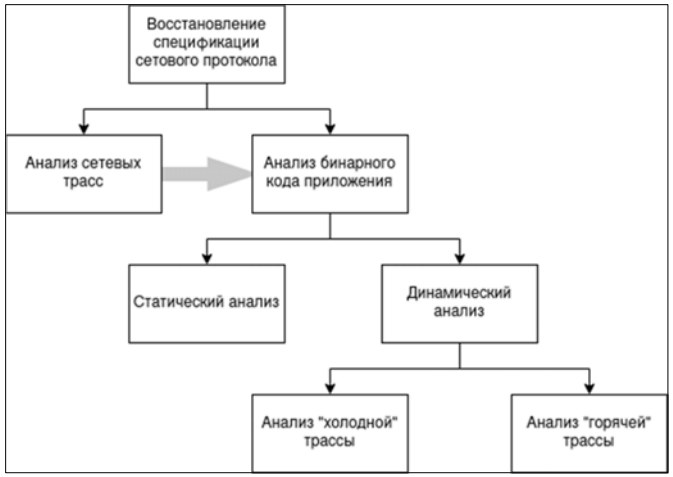

Рис. 1. Способы восстановления спецификаций протоколов и протокольных автоматов
Fig. 1. Ways for recovering protocol specifications and protocol automata

Использование записанных сетевых трасс в качестве начальных данных для восстановления протокольного автомата позволяет решать данную задачу в условиях, когда невозможно получить доступ к бинарному коду программы, при этом применяются, как аналитические подходы, так и автоматизированные алгоритмы, основанные на статистических методах.

Анализ бинарного кода приложения методом изучения записанных («холодных») трасс хорошо автоматизируется, позволяет применять технологии динамического символьного выполнения, восстанавливать алгоритмы.

Применение методов динамической инструментации программ позволяет исследовать «горячую» трассу, т. е. изучать бинарный код программы в ходе ее выполнения. Такой способ восстановления протокольного автомата из бинарного кода – это своего рода фаззинг, но в отличие от «классического» понимания фаззинга здесь основной задачей выступает

восстановление закономерностей и всех возможных последовательностей сообщений [1], а не подбор входных данных, вызывающих некорректное поведение программы (аварийное завершение, зависание и т. п.), другими словами, объектом изучения подобных исследований является не состояние программы, а состояние протокола. Дальнейшее повествование статьи посвящено именно этой проблематике.

Задача реинжиниринга спецификации протокола состоит из восстановления форматов данных и менее изученного вопроса – восстановления реализованного в программе протокольного автомата. Автомат связан с внутренним состоянием сетевой программы, переход из одного состояния в другое выполняется при получении пакета входных данных определенного формата. Для восстановления понимания работы программы необходимо уяснить, что именно определяет состояние протокола и по каким правилам происходят переходы. Кроме того, после решения указанной задачи необходимо выполнить тестирование на предмет определения насколько корректно и полно восстановлен автомат. В статье предлагаются методы, позволяющие автоматизировано решать эти задачи.

2. Обзор научно-технической информации

В настоящее время сложились два основных подхода к восстановлению протокольных автоматов сетевых протоколов, базирующиеся на использовании различных начальных данных: на основе анализа записанных сетевых трасс и путем анализа трассы выполнения программы, реализующей исследуемый протокол. Оба указанных способа имеют право на существование и позволяют решать поставленную задачу в различных начальных условиях.

Например, при осуществлении исследований проприетарных промышленных протоколов, используемых для автоматизации управления производственными процессами, объектов социальной или критической инфраструктуры в условиях отсутствия доступа к исполняемым файлам и исходным текстам решение задачи возможно только путем анализа записанных сетевых трасс.

В других случаях, когда в ходе решения задачи существует возможность запустить бинарный код, реализующий протокол, в контролируемом окружении, для исследования можно применять информацию, полученную в ходе анализа трасс выполнения программ или программно-аппаратных комплексов.

В обзорной статье венгерских исследователей [4], рассмотрены 39 инструментов и технологий реверс-инжиниринга сетевых протоколов и восстановления протокольных автоматов, информация о которых была представлена в свободном доступе в период с 2005 по 2017 гг. Среди них только 13 позволяют восстанавливать протокольные автоматы. Аналогичный обзор, представленный в источнике [5] в начале 2021 года, позволяет расширить список результатами, полученными после 2017 года, среди которых, в контексте задачи восстановления протокольных автоматов, можно выделить 2 работы [6, 7].

Наиболее свежие результаты были представлены в августе 2021 года в статье [2], при этом применяется метод восстановления форматов сообщений, описанный в работе [8] тех же авторов.

Обобщенные сведения о результатах указанных работ и возможностях созданных инструментов приведены в табл. 1.

Табл. 1. Инструменты восстановления протокольных автоматов
Table 1. List of protocol automata reverse engineering tools for network protocols

Название/авторы	Год	Практические результаты			Вход		Выход	
		Т. пр.	**Б. пр.**	**Др. пр.**	**СТ**	**ТВ**	**ФП**	**ПА**
GAPA [9]	2005	HTTP				✓	✓	✓
PEXT [3]	2007	FTP			✓	✓		✓
Prospex [10]	2009	SMTP, SIP	SMB	Agobot (C&C)	✓	✓	✓	✓
Xiao и др. [11]	2009	HTTP, FTP, SMTP				✓		✓
Trifilo и др. [12]	2009		TCP, ARP, DHCP, KAD		✓			✓
Antunes and Neves [13]	2009	FTP			✓			✓
ReverX [14]	2011	FTP			✓		✓	✓
Veritas [15]	2011	SMTP	PPLIVE, XUNLEI		✓			✓
Biprominer [16]	2011		XUNLEI, QQLive, SopCast		✓		✓	✓
Zhang и др. [17]	2012	HTTP, SNMP, ISAKMP			✓			✓
Netzob [18, 19]	2012	FTP, Samba	SMB	P2P? VoIP?	✓		✓	✓
AutoReEngine [20]	2013	FTP	DHCP		✓			✓
Laroche и др. [21]	2013	HTTP, FTP, SMTP, POP3	DNS, NetBIOS		✓		✓	✓
Meng и др. [22]	2014		TCP, ARP		✓			✓
PREUGI [6]	2017				✓			✓
Goo и др. [7]	2019	HTTP	DNS		✓		✓	✓
Gábor Székely и др. [2]	2021				✓		✓	✓

т.пр. – текстовые протоколы, **б.пр.** – бинарные протоколы, **др.пр.** – другие неизвестные протоколы, **СТ** – сетевая трасса, **ТВ** – трасса выполнения программы, **ФП** – формат протокола, **ПА** – протокольный автомат

Из приведенной таблицы видно, что в большинстве случаев восстановление протокольных автоматов основано на анализе ранее записанных сетевых трасс. Основная проблема при таком подходе – это восстановление форматов сообщений. Формат сообщения, это набор полей и их объединение в структуры и последовательности, а также информация о семантике полей – их роли в структуре сообщения. В условиях отсутствия знаний об алгоритме обработки сообщения восстановление информации о его составляющих элементах и их семантике с достаточной точностью является весьма сложной задачей. Для ее решения используются различные эвристические, статистические [15], грамматические [6, 17] алгоритмы, методы анализа последовательностей [23] и т.д. Наибольшие трудности, очевидно, возникают при анализе бинарных протоколов. Кроме того, очевидно, что в случаях применения кодирования или криптографических методов защиты передаваемых данных задача восстановления форматов сообщений из сетевой трассы становится, в лучшем случае, трудноразрешимой.

В подходе [2] для описания протокольного автомата используется автомат Мили, отличающийся от обычного конечного автомата тем, что для каждого перехода, инициированного входным сигналом, определен выходной сигнал. В предложенном методе применяются модифицированный алгоритм L_M+ [5] построения автомата по принципу обучения и алгоритм минимизации автомата [24]. Однако в данной работе не рассматриваются подходы к восстановлению форматов и классификации сообщений, предполагается, что эта задача уже решена.

Способы восстановления протокольных автоматов на основе записи и дальнейшего анализа трасс выполнения целевого бинарного кода в приведенной таблице датируются более чем десятилетней давностью. С точки зрения функциональных возможностей среди указанных работ наибольший интерес представляет Prospex [10]. В данном подходе для получения начальных данных требуется пара приложений клиент и сервер. Интересующие программы запускаются на выполнение в контролируемом окружении, при этом осуществляется анализ потока данных при обработке входных сообщений, в результате восстанавливаются форматы сообщений. Далее осуществляется анализ сетевых трасс, анализируются цепочки обмена данными, а сообщения классифицируются с использованием алгоритма анализа последовательностей [23], после чего в соответствии с восстановленными форматами вырабатываются обобщенные форматы. На последнем этапе, используя имеющиеся цепочки обмена сообщениями и обобщенные форматы, с помощью эвристических алгоритмов строится протокольный автомат, работа завершается минимизаций автомата с помощью алгоритма Exbar [25]. Ограничением в данной работе выступает отсутствие возможности работы с клиентскими приложениями. Кроме того, в открытом доступе не представлено исходных текстов инструмента Prospex, а исследователи более не публиковали какие-либо дополнительные результаты по данной тематике. Вместе с тем, предложенный подход выглядит перспективным и будет использован в качестве отправной точки для создания метода восстановления протокольных автоматов сетевых протоколов.

3. Математическая модель протокольного автомата

3.1 Общее описание

Существуют два подхода к формальному описанию процессов, функционирующих в режиме «запрос-ответ», с помощью математического аппарата конечных автоматов [26].

1) За основу берутся состояния (protocol based/protocolar). В данном случае в качестве вершин графа выступают состояния протокола, а ребра описывают обобщенные действия и задают функцию переходов из состояния в состояние, выполняемых под воздействием внешних сигналов. Этот способ описания позволяет строить абстрактные протокольные автоматы.
2) За основу берутся действия или поведение (behavior based/behavioral). В качестве вершин определяются действия системы, в результате выполнения которых вырабатывается сигнал, а ребра описывают этот сигнал и определяют функцию переходов. Такой подход позволяет описывать управляющие автоматы. Подобного рода автоматы показывают связь между состояниями программы и состояниями протокола.

В статье для описания протокольных автоматов используется первый поход, однако, стоит отметь, что используется аппарат не классических конечных автоматов, а, скорее, его расширение, допускающее использованием некоторого набора переменных, влияющих на поведение. Кроме того, необходимо обратить внимание, что этот подход применим для описания систем, осуществляющих взаимодействие в режиме реального времени, а не в отложенном режиме.

Рассмотрим основные понятия, необходимые для дальнейших рассуждений.

Под *системой* будем понимать программное средство осуществляющее взаимодействие с использованием сетевого интерфейса.

Протокол – это формально описанный абстрактный язык взаимодействия/коммуникации систем.

Системы будем называть *однородными*, если они могут осуществлять длительное результативное взаимодействие с использованием одного и того же протокола, т. е. системы при выполнении запросов получают релевантные ответы.

Передаваемые в рамках взаимодействия однородных систем данные формируются на основе грамматических и синтаксических правил протокола и называются *сообщениями*.

Входящее (входное) сообщение – это сообщение, поступившее в систему, исходящее (выходное) сообщение – отправленное системой.

В случаях, когда хотя бы одно сообщение зависит от одного или нескольких предыдущих входных и/или выходных, протокол называется протоколом чувствительным к состояниям (stateful-протокол), в противном случае – протоколом не чувствительным к состояниям (stateless-протокол).

В ходе проведения исследований в изученных материалах не было найдено определения понятия *состояние протокола*, которое можно однозначного формализовать и описать с использованием элементов математических теорий, в силу чего определим его следующим образом.

Состояние протокола – это состояние всех *глобальных объектов* (некоторых данных, значения которых определяются в процессе взаимодействия однородных систем и используются до завершения их коммуникации), оказывающих влияние на порядок следования и содержимое входных и выходных сообщений. Исходя из приведенного определения, stateless-протоколы не имеют глобальных объектов, чем и отличаются от stateful-протоколов.

Начальное состояние протокола – состояние, в котором система ожидает получения (серверная часть) или отправки (клиентская часть) первого сообщения.

Конечное (терминальное) состояние протокола – состояние, в котором невозможно дальнейшее получение входных и отправка выходных сообщений в рамках данного сеанса сетевого взаимодействия. Важно отметить, что в отдельных случаях конечное состояние может отсутствовать, например при реализациях промышленных протоколов, нацеленных на длительное непрерывное функционирование, например, в IoT-устройствах.

Смена состояния протокола осуществляется под воздействием входных сообщений (сигналов), при этом индикатором перехода в следующее состояние является отправка выходного сообщения или закрытие канала связи.

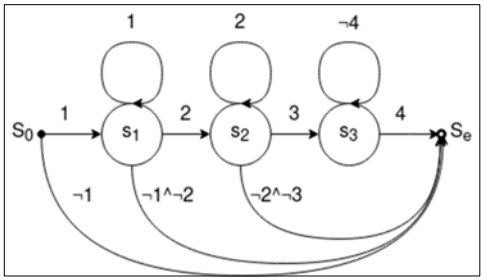

Рис. 2. Пример простого графа протокольного автомата
Fig. 2. Simple protocol automata graph sample

Протокольный автомат – это абстрактная математическая модель, предназначенная для формального описания протокола. Для наглядности протокольный автомат удобно представлять в виде ориентированного графа переходов, в качестве вершин которого

выступают состояния протокола, а ребра задают *функцию перехода* из состояния в состояние и помечаются входными и, при необходимости, выходными сигналами (рис. 2).

	1	*2*	*3*	*4*	*др.*
S_0	S_1	S_e	S_e	S_e	S_e
S_1	S_1	S_2	S_e	S_e	S_e
S_2	S_e	S_2	S_3	S_e	S_e
S_3	S_3	S_3	S_3	S_e	S_3
S_e	–	–	–	–	–

Рис. 3. Описание протокольного автомата с помощью таблицы
Fig. 3. Table representation of the defined early protocol automata

Кроме графического представления, используется табличная (матричная) запись автоматов. Каждой строке такой таблицы соответствует одно состояние, а столбцу входной сигнал. В ячейке на пересечении строки и столбца записывается следующее состояние и, если требуется, выходной сигнал. Табличное представление автомата, показанного на рис. 2, приведено ниже (рис. 3). Данный способ удобен с точки зрения программной реализации протокольного автомата.

Обобщая изложенное, зададим протокольный автомат *M* шестеркой объектов:

$$\mathrm{M} = (S, P, Q, f, s_0, s_e),$$

где:

S: $|S| < \infty$ – множество состояний протокола;

P: $|P| < \infty$ – множество входных сигналов;

Q: $|Q| < \infty$ – множество выходных сигналов;

f: $S \times P \times Q \to S$ – функция переходов из состояния в состояние;

$s_0 \in S$ – начальное состояние;

$s_e \in S$ – конечное состояние (в отдельных случаях конечное состояние может отсутствовать).

Множество состояний протокола *S*, а также множества входных *P* и выходных сигналов *Q* конечны. Функция $f: S \times P \times Q \to S$ осуществляет переход в следующее состояние, при этом оценивается текущее состояние протокола, анализируется входной сигнал и формируется выходной сигнал для ответной части. При этом для любого состояния протокола (кроме конечного) существует входной сигнал, переводящий его в другое, то же самое состояние или конечное состояние, и соответствующий этому выходной сигнал:

$$\forall s \in S : s \neq s_e, \exists p \in P, \exists q \in Q \; u \; \exists s' \in S : f(s,p,q) = s'.$$

3.2 Проецирование математической модели на бинарный код

Для построения протокольного автомата сетевого протокола компьютерной программы необходимо спроецировать построенную математическую модель на бинарный код, для чего требуется решить следующие задачи:

- определить множество состояний *S*, зависящее от некоторых глобальных объектов и стека вызовов, выделить начальное и конечное состояния (может отсутствовать);
- построить множества *P* и *Q* входных и выходных сообщений, соответственно, сформировать возможные пары *(p, q)* из множества $P \times Q$;
- построить функцию переходов;
- определить эквивалентные состояния и минимизировать протокольный автомат.

Как было определено ранее, состояние протокола характеризуется состоянием неких глобальных объектов, по значениям которых или их совокупности можно судить о различии состояний протокола и выполнении перехода из одного в другое. Таким образом, для

построения множества состояний необходимо определить набор глобальных объектов и обеспечить возможность контроля и сравнения их значений на различных этапах выполнения программы.

Выполнившийся алгоритм можно представить в виде линейно упорядоченного списка вызванных инструкций. Линейное представление бинарного кода позволяет определить зависимости одних данных от других. В частности, если на некотором шаге *i* полученные данные были каким-либо образом обработаны и сохранены (на стеке или в куче), а на каком-то из следующих шагов $j > i$ они были использованы в процессе обработки вновь полученных данных или подготовки выходного сообщения, тогда, очевидно, что новые данные зависят от полученных ранее при условии, что в интервале шагов *(i, j)* интересующие нас блоки данных в адресном пространстве не перезаписывались. В состав глобальных объектов введем множество S_{num} – порядковых номеров инструкций в трассе выполнения программы.

Реализация протокольных автоматов в программном коде может быть выполнена различными способами, в том числе крайне нестандартными. Вместе с тем, в ходе изучения исходных текстов сетевых программ, например, СУБД Postgres, FTP-сервера ioFTPD и др., а также, исходя из личного опыта разработки, были выделены четыре наиболее часто встречаемых. Рассмотрим их на примере протокольного автомата, представленного в виде графа на рис. 5.

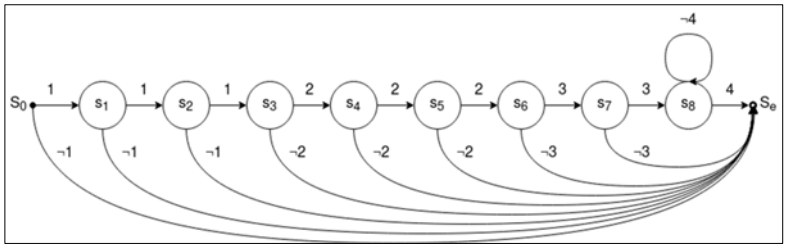

Рис. 5. Пример протокольного автомата для дальнейших рассуждений
Fig.5. Protocol automata sample for the further reasoning

1) Линейная последовательность вызовов функции приема и выполнения анализа входных сообщений, приводящая к конечному циклу обработки. С-подобный псевдокод приведен в листинге 1.

```
…
recv(…buf…);
if(buf != 1)cleanup_and_exit();
recv(…buf…);
if(buf != 1)cleanup_and_exit();
recv(…buf…);
if(buf != 1)cleanup_and_exit();
recv(…buf…);
if(buf != 2)cleanup_and_exit();
while(1) {
  recv(…buf…);
  if(buf == 4) cleanup_and_exit();
}
…
```

Листинг 1. С-подобный псевдокод линейного алгоритма
Listing 1. C-like pseudocode of the linear algorithm

В данном случае легко видеть, что смена состояния протокола будет равносильна смене адреса инструкции вызова функции получения входных данных, а в общем случае, если

для приема входных данных реализована отдельная функция, тогда различающимся состояниям соответствуют различающиеся стеки вызовов, завершающиеся вызовом системной функции `recv(…)`. Будем рассматривать это утверждение в качестве критерия, позволяющего судить о смене состояния в ходе анализа бинарного кода, а стек вызовов до вызова функции получения сообщения в качестве одного из глобальных объектов протокола.

2) Последовательный набор циклов (листинг 2).

```
int i = 0;
while(i<3) {
  recv(…buf…); i++;
  if(buf != 1) cleanup_and_exit();
}
i = 0;
while(i<3) {
  recv(…buf…); i++;
  if(buf != 2) cleanup_and_exit();
}
i = 0;
while(i<2) {
  recv(…buf…); i++;
  if(buf != 3) cleanup_and_exit();
}
while(1) {
  recv(…buf…);
  if(buf == 4) cleanup_and_exit();
}
```

Листинг 2. С-подобный псевдокод алгоритма последовательных наборов циклов
Listing 2. Algorithm C-like pseudocode of the sequential cycles

Применяя критерий смены состояния, сможем построить автомат, показанный на рис. 6. Нетрудно видеть, что изначальный автомат не будет эквивалентен полученному, но будет его частным случаем.

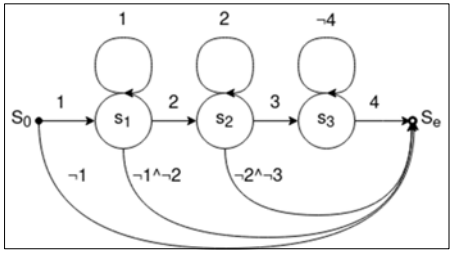

Рис. 6. Автомат, построенный по предложенному коду с использованием первого критерия
Fig. 6. Construction of protocol automata based on the code sample using the first criteria

С точки зрения состояний программы состояния протокола будут описываться значением переменной *i* на конкретном шаге алгоритма. Однако привязать эту переменную к обработке пакетов при анализе трассы выполнения весьма затруднительно, но, проследив количество выполнений одной и той же инструкции вызова функции получения входных данных, можно конкретизировать полученный автомат. Таким образом, в качестве одного из глобальных объектов протокола можно рассматривать счетчик количества выполнения инструкции вызова функции получения данных, сопровождающихся

последующей отправкой выходного сообщения и соответствующих одному и тому же стеку вызовов.

3) Единый цикл получения и анализа данных с блоком условных переходов (листинг 3).

```
prot_state cur_state = INIT_STATE;
while(curr_state != END_STATE) {
  recv(…buf…);
  switch(buf) {
    case 1: … break;
    case 2: … break;
    case 3: … break;
    case 4: … break;
    default: … break;
  }
}
```

Листинг 3. С-подобный псевдокод цикла с блоком условных переходов
Listing 3. C-like pseudocode of a single cycle with conditional branch's block

Очевидно, что, в данном случае, при обработке очередного пакета в зависимости от содержимого сообщений или вычисляемых на их основе значений переменных граф потока управления меняется.

Для различия сетевых сообщений одного протокола используют различные контрольные значения их содержимого: заголовки, команды, служебные данные и т. п., - назовем их ключевые поля.

Ключевые поля сообщений строго определены. Их обработка обеспечивает возможность выбирать конкретное направление развития алгоритма, формирует правила принятия решений о смене состояния протокола. В бинарном коде эта задача решается с помощью инструкций условных переходов, предназначенных для передачи управления одной из заданных инструкций. Таким образом, описанный ранее линейный список выполненных инструкций представляется в виде графа потока управления, вершинами которого выступают адреса инструкций, а ребра представляют собой передачу управления от одной инструкции к другой. Сообщения будут равны тогда и только тогда, когда графы потока управления, сформированные в результате работы алгоритма их обработки, равны.

Очевидно, что при различных значениях ключевых полей, после их сравнения с референтными значениями, во время выполнения инструкций условных переходов будут выбираться различные пути развития алгоритма обработки сообщений, в силу чего и сформированные графы потока управления будут различаться.

Другими словами, вектора, составленные из адресов пройденных базовых блоков, будут различными при различных состояниях протокола. Совокупность таких векторов можно ввести в состав глобальных объектов протокола.

4) Единый цикл получения и обработки данных, где в зависимости от полученного сообщения изменяется указатель на функцию обработчик (листинг 4).

```
typedef void* (*process_func)(void*);
typedef struct _state_ {
  byte id;
  process_func process_message;
} prot_state;
prot_state cur_state(INIT_STATE);
while(curr_state != END_STATE) {
  recv(…buf…);
```

```
    cur_state =

  (prot_state*)state[cur_state].process_message(buf);
  }
```

Листинг 4. С-подобный псевдокод цикла с изменяющейся функцией-обработчиком сообщения
Listing 4. C-like pseudocode of a single cycle with a variable processing function

В данном примере очевидно, что с использованием первого критерия все полученные сообщения будут отнесены к одному и тому же состоянию, однако состояние программы в ходе ее выполнения будет меняться, в частности переменная *cur_state*, значение которой зависит от обработанных входных данных, и указатель на вызываемую функцию-обработчик. Таким образом, можно утверждать, что если в ходе выполнения программы в процессе обработки очередного входного сообщения в паре:

$$\langle addr_{call\_instr}, addr_{callee\_fn} \rangle,$$

где:

$addr_{call\_instr}$ – адрес инструкции вызова;

$addr_{callee\_fn}$ – адрес вызываемой функции;

изменяется значение $addr_{callee\_fn}$ – адреса функции, при одном и том же $addr_{call\_instr}$ – адресе инструкции вызова, тогда можно утверждать, что протокол сменил состояние, т.е. совокупность глобальных объектов протокола можно дополнить парами указанного вида.

Обобщая изложенное, исходя из сформулированного определения состояния протокола, в качестве глобальных объектов протокола будем рассматривать сохраненные в адресных пространствах данные, доступные в рамках работы алгоритмов протокола, а также адреса инструкций, вызывающих функции (системные вызовы) получения сообщений и адреса самих функций.

Введем обозначения:

V_{taint} – множество всех переменных и блоков в адресных пространствах, «порожденных» в ходе анализа потока данных при обработке входных сообщений, элементы множества перезаписываемые.

I_{recv} – множество всех векторов, содержащих стек вызовов до вызова системной функции получения входных сообщений.

C_{proc} – множество, включающее заключенные в интервале между шагами получения и отправки сообщений, всех пар вида:

$$\langle addr_{call\_instr}, addr_{callee\_fn} \rangle,$$

где $addr_{call\_instr}$ – адрес инструкции вызова, $addr_{callee\_fn}$ – адрес вызываемой функции.

S_{num} – множество порядковых номеров выполнившихся инструкций.

$O_G = V_{taint} \times I_{recv} \times C_{proc} \times S_{num}$ – множество возможных наборов глобальных объектов протокола.

Описанный подход позволяет сформировать множество различающихся состояний изучаемого протокола, кроме того, можно утверждать, что существует некоторая функция $F: O_G \to S$, отображающая состояние множества глобальных объектов в множество состояний протокола. Функция F позволяет описать все возможные состояния. Очевидно, что функция F сюръективна, т.е. $\forall s \in S \, \exists o \in OG : F(o) = s$. Если предположить, что это не так, тогда в протоколе будет недостижимое состояние, другими словами, не являющееся состоянием протокола. Кроме того, это позволяет сделать предположения, что в процессе создания программного кода была допущена ошибка.

Множества P и Q – входных и выходных сообщений можно сформировать, осуществив перехват параметров функций приема и отправки сигналов.

Отметим связи между автоматами и некоторыми событиями потока управления, как например, выполнение функции `accept(…)` и `connect(…)` позволяет сделать вывод о выходе протокола из начального состояния и начале взаимодействия, `recv(…)` – получение входного сигнала, `send(…)` – потенциально, может служить индикатором перехода в следующее состояние, а `close(…)` – однозначно указывает на прекращение соединения и переход протокольного автомата в конечное состояние. Для построения функции переходов будем отталкиваться от описанных связей бинарного кода и математической модели. Кроме того, будем считать, что если один и тот же входной сигнал был успешно обработан более одного раза подряд и при этом были сформированы и отправлены одинаковые выходные сообщения, тогда протокольный автомат переходит в то же самое состояние, т. е. формируется петля в графе.

4. Описание разработанных макетов инструментов

Описанный в разд. 2 подход позволяет построить протокольный автомат, реализованный в бинарном коде, предполагающий штатное функционирование системы, однако наиболее интересной задачей представляется выявление дополнительных (сервисных) состояний, ошибок и недекларированных возможностей протокола, а также дефектов тестируемого бинарного кода.

Фаззинг протокола заключается в автоматической генерации входных сигналов, передаче их для обработки программе и последующей оценке состояния протокольного автомата.

В результате выполнения теста может либо осуществиться переход автомата из одного состояния в другое, либо нет, при этом если тест позволяет увеличить карту покрытия кода, обеспечивающего обработку входных данных, тогда этот тест также представляет интерес поскольку гарантирует получение сообщения отличного от предыдущих, что позволяет выявлять дефекты бинарного кода и уточнять грамматику для порождения входных сигналов.

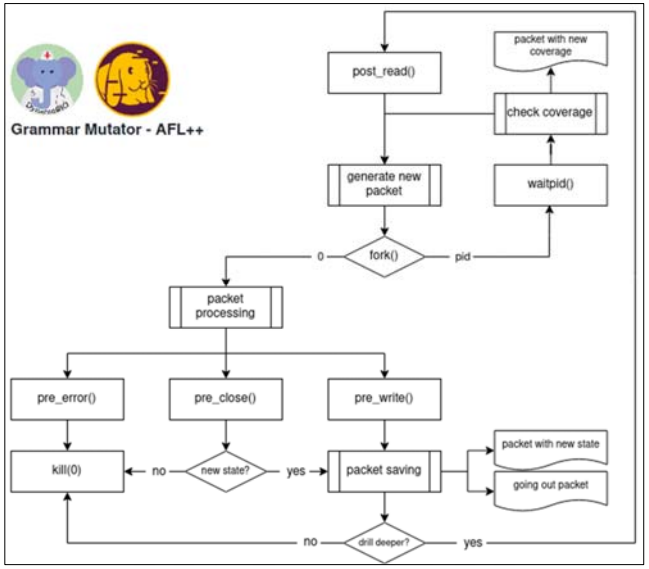

Рис. 7. Подход к восстановлению протокольного автомата сетевой программы (общая схема)
Fig. 7. Approach to recover protocol automata of a network application (common scheme)

В рамках реализации макета инструмента выделены три режима фаззинга протокола, в зависимости от решаемой задачи:

- фаззинг текущего состояний с целью увеличения покрытия;
- фаззинг с целью выполнения максимально возможного количества переходов автомата;
- комбинированный подход.

Общая схема реализованного подхода к фаззингу протокола представлена на рис. 7.

Суть метода заключается в создании динамической/плавающей точки запуска порождающего сервера (форк-сервера), обеспечивающего многократное повторения алгоритма на разных входных данных с учетом состояний протокола.

В качестве точки порождения выбирается инструкция вызова функции (системного вызова) получения входного сигнала, после чего генерируются новые данные, подменяются значения аргументов и возвращаемого значения функции (системного вызова) и анализируется процесс их обработки, включая граф потока управления и граф потока данных. Если тест заканчивается переходом на следующее состояние, тогда в зависимости от опций запуска либо осуществляется переход и выполняется порождение очередного потомка, либо оценивается покрытие, сохраняется результат и повторяется исходный тест.

Преимущество такого подхода – осуществление фаззинга непосредственно в памяти тестируемого процесса (in-memory fuzzing).

В данном контексте возникает три задачи:

- инструментация целевой программы для обеспечения возможности внедрения в ее код требуемых инструкций и выполнения «горячего» анализа потока данных и потока управления.
- генерация грамматически верных данных для уменьшения количества «пустых» тестов;
- оценка карты покрытия.

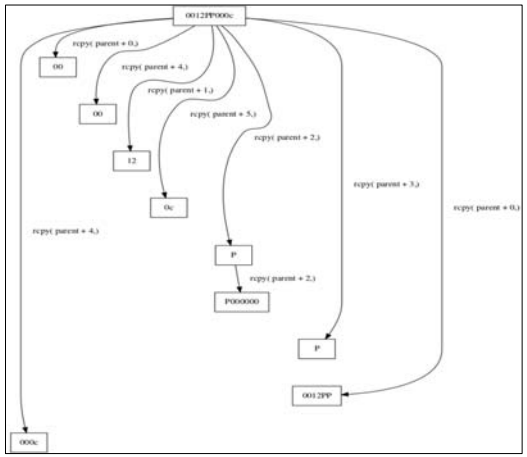

Рис. 8. Результат автоматического восстановления формата входного сообщения
Fig. 8. Automatic input signal format restoring result (graphical representation)

Для динамической инструментации при разработке инструментов выбрана система DynamoRIO – удобный инструмент для разработки инструментов анализа бинарного кода приложений, работающих в пользовательских режимах, DynamoRIO предназначена для функционирования в среде операционных систем (ОС) семейств Windows, Linux, Android,

экспериментальная версия поддерживает Mac. Допустимые архитектуры процессоров: IA-32, AMD64, ARM и AArch64.

Для изучения форматов входных сообщений разработан макет инструмента также с использованием системы DynamoRIO. Аналогично подходу, реализованному в инструменте Vigilante [27], алгоритм анализа потока данных, основан на инструментации инструкций передачи управления, чтения и записи данных адресных пространств, а также обертывании стандартных вызовов, используемых для работы с памятью, сокетами и функций криптографической библиотеки OpenSSL. Пример результата работы инструмента показан на рис. 8.

Генерация грамматически верных данных обеспечивается с помощью инструмента Grammar Mutator [23] из набора инструментов AFL++ [28]. В настоящее время грамматики описываются вручную на основе блоков входных сообщений, выделенных в процессе анализа их форматов.

Оценка покрытия выполняется методом, реализованным в фаззере AFL++.

Обобщенно реализованный подход описывается следующими шагами:

1) инструментируем целевую программу, запускаем, осуществляем взаимодействие с ответной частью системы, в результате формируется множество $P \times Q$, получаем описание форматов входных сообщений и протокольного автомата;
2) описываем грамматики входных сообщений, собираем библиотеки для генерации данных;
3) запускаем стенд, сохраняем максимально возможное количество пар из множества $P \times Q$, формируем начальную матрицу переходов, все непроверенные переходы заполняет нулями;
4) проводим эксперимент по отправке сообщений в произвольном порядке, уточняем матрицу;
5) запускаем процесс фаззинга протокола с использованием механизмов генерации данных на основе сформированных грамматик и учета покрытия кода, фиксируем новые пары из множества $P \times Q$, заполняем ранее нулевые поля, при необходимости расширяем матрицу, выявляем эквивалентные состояния, редуцируем автомат.

5. Описание экспериментов с VPN-клиентом

Цель экспериментов заключалась в оценке работоспособности предложенного метода на примере разработанных макетов инструментов.

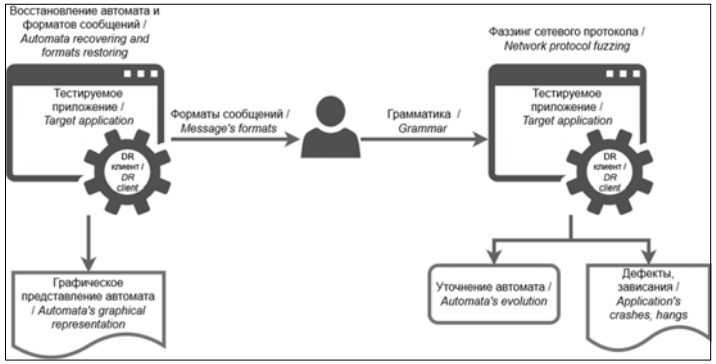

Рис. 9. Общая схема применения разработанных макетов инструментов
Fig. 9. Common scheme for using the developed tools

Задачи экспериментов:

- восстановление протокольного автомата, последовательностей и форматов сообщений;
- тестирование протокольного автомата методом фаззинга.

Для проведения верифицируемых экспериментов в качестве объекта был выбран VPN-клиент с открытыми исходными текстами, реализующий технологию защищенного SSL VPN туннеля. Общая схема применения разработанных макетов инструментов в рамках проведенных экспериментов показана на рис. 9.

На начальном этапе тестируемое приложение запускалось с целью восстановления форматов и последовательностей сообщений, а также протокольного автомата с помощью предназначенного для этого разработанного макета инструмента. В результате выделены три фазы сетевого функционирования клиента:

- процедура SSL-handshake, реализуемая с помощью библиотеки OpenSSL (анализ не проводился);
- авторизация на сервере, выполняемая с помощью протокола HTTP;
- функционирование канала на основе протокола PPP.

Для фаз подключения, в рамках которых осуществлялось взаимодействие с использованием протоколов HTTP и PPP, записаны последовательности входных и исходящих сообщений, а также в формате DOT созданы деревья разбора входных сообщений. Кроме того, получены данные для восстановления протокольного автомата. Графическое представление восстановленного автомата представлено на рис. 10, HTTP- и PPP-фазы отображены прямоугольниками с соответствующими текстовыми метками.

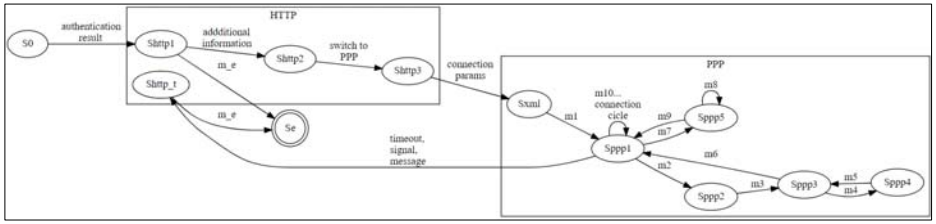

Рис. 10. Восстановленный протокольный автомат VPN-клиента (графическое представление)
Fig. 10. VPN-client's restored protocol automata (graphical representation)

На втором этапе экспериментов было выполнено изучение деревьев разбора входных сообщений (осуществлялось в ручном режиме), выделены ключевые поля, в результате разработана грамматика и инструмент, позволяющий генерировать псевдо HTTP-код (листинг 5).

```
{
  "<START_LINE>": [["HTTP/1.1 200 OK\r\n", "<HEADERS>", "\r\n\r\n",
                    "<BODY>", "\r\n\r\n"]],
  "<HEADERS>": [[], ["<HEADER>", "<DELIM>", "<HEADERS>"]],
  "<DELIM>": [[], ["\r"], ["\n"], ["\r\n"], ["\n\r"], ["\u0000"]],
  "<HEADER>": [["<HEADER_FIELD>", ": ", "<ANY>"]],
  "<HEADER_FIELD>": [["Content-Length"], ["Set-Cookie"]],
  "<URI>": [["<SCHEME>" , ":", "<HIER>", "<FRAGMENT>"]],
  "<SCHEME>": [["http"], ["https"], ["shttp"], ["dav"], ["attachment"],
               ["data"], ["file"], ["ftp"]],
  "<HIER>": [["//", "<AUTHORITY>", "<PATH>"]],
  "<AUTHORITY>": [["<USERINFO>", "<HOST>"]],
  "<PATH>": [["/", "<DIR>"]],
  "<DIR>": [[], ["<CHARS>", "/", "<DIR>"]],
  "<FRAGMENT>": [[], ["#", "<CHARS>"]],
  "<BODY>": [[], ["<CHARS>"]],
  "<SVPN>": [["SVPNCOOKIE="]],
```



```
  "<SVPNCOOKIE>": [["<SVPN>", "<CHARS>"]],
  "<ANY>": [[], ["<CHAR>"], ["<CHARS>"], ["<HOST>"], ["<URI>"],
            ["<SVPNCOOKIE>"]],
  "<CHARS>": [[],["<CHAR>", "<CHARS>", "<CHARS>", "<CHARS>"]],
  "<CHAR>": [["0"], ["1"], ["2"], ["3"], ["4"], ["5"], ["6"], ["7"],
             ["8"],
          ["9"], ["a"], ["b"], ["c"], ["d"], ["e"], ["f"], ["g"], ["h"],
          ["r"], ["s"], ["t"], ["u"], ["v"], ["w"], ["x"], ["y"], ["z"],
          ["="]]
}
```

Листинг 5. Грамматика для генерации входных данных в ходе фаззинга
Listing 5. Grammar for generating fuzzing input data

На третьем этапе экспериментов с использованием разработанного макета инструмента фаззинга протоколов, выполнено тестирование HTTP-фазы автомата. С использованием сформированной грамматики выполнялась генерация псевдо HTTP-кода, который использовался в качестве входных данных в процессе фаззинга.

В результате построена последовательность входных сообщений, позволяющая перейти из начального состояния *S0* к состоянию *Sppp1* (табл. 2). Кроме того, обнаружены сообщения, приводящие к зависанию VPN-клиента в состояниях *Shttp1*, *Shttp2*, *Shttp3*.

Табл. 2. Восстановленная последовательность входных сообщений
Table. 2. Recovered input sequence leading to Sppp1 state

Состояние	**Описание входного сообщения от сервера**	**Содержимое сгенерированного с помощью грамматики входного сообщения, позволяющего перейти к следующему состоянию**
Shttp1	HTTP сообщение размером 782 байта, содержащее заголовки и html-контент	HTTP/1.1 200 OK Set-Cookie: SVPNCOOKIE=
Shttp2	HTTP сообщение размером 2105 байт, содержащее заголовки и html-контент	HTTP/1.1 200 OK
Shttp3	HTTP сообщение размером 2099 байт, содержащее заголовки и html-контент	HTTP/1.1 200 OK
Sxml	HTTP сообщение размером 914 байт, содержащее заголовки и xml-контент	HTTP/1.1 200 OK Set-Cookie: ssh://<USERINFO><HOST>/ Content-Length: Content-Length: t
Sppp1	…	…

Анализ исходных текстов тестируемой программы показал, что зависания происходят из-за того, что разработчиками не предусмотрена возможность получения сообщения с некорректным содержимым. Предполагается, что все сообщения будут удовлетворять форматам данных, отправляемых сервером, в противном случае алгоритм считает, что получена только часть сообщения и переходит к ожиданию оставшихся данных и зависает на функции чтения из сети.

6. Преимущества подхода, ограничения и перспективы

Предлагаемый подход позволяет в короткое время в автоматизированном режиме построить протокольный автомат сетевого протокола, сгенерировать большой корпус данных, обеспечивающих переход автомата из состояния в состояние и позволяющих увеличивать покрытие кода, реализующего алгоритм протокола.

Анализ трассы выполнения программы позволяет заглянуть внутрь процесса инкапсуляции протоколов, проанализировать содержимое сигналов, которые зашифрованы или

закодированы при передаче по каналу связи, кроме того, в отличие от подходов основанных на анализе сетевых трасс, имеющих сложности при исследовании протоколов без обратной связи, данный подход позволяет в ходе анализа потока управления, оценивая результаты выполнения сетевых функций, однозначно идентифицировать некоторые состояния протокола.

На момент написания статьи на использование разработанных макетов накладываются ограничения в возможности исследования приложений, предназначенных для функционирования в ОС семейства Linux, на архитектуре процессора x86_64. Дополнительные ограничения в силу отсутствия достаточной статистики применения пока не определены.

В настоящее время выполняется решение следующих перспективных задач.

1) Разработка механизма для автоматического комбинированного фаззинга, позволяющего одновременно тестировать текущие состояния и выполнять переходы к следующим.
2) Разработка механизма использования ролевых моделей функций для обеспечения возможности управления алгоритмом инструментирования тестируемой программы без необходимости модификации исходных текстов.
3) Автоматизированное формирование скриптов с описанием начального протокольного автомата и формата входных сигналов, а также расширение способов генерации данных в ходе фаззинга протоколов.
4) Разработка способа формального описания протокольных автоматов.
5) Автоматизированное создание исполняемого файла ответной части исследуемой программы, обеспечивающей возможность воспроизведения сгенерированных последовательностей сообщений и выявленных ошибок реализации протокола, а также дефектов бинарного кода.
6) Создание версий, совместимой с архитектурой ARM в ОС Linux и предназначенной для использования в среде ОС Windows.
7) Расширение набора протестированных программ.

7. Заключение

В статье описан подход к восстановлению протокольных автоматов по бинарному коду сетевых программ и фаззингу протоколов, при этом разработаны:

- автоматизированный эвристический подход к определению типа протокола (о других существующих методах решения этой задачи автору ничего не известно), выявлению состояний и построению протокольного автомата в «первом приближении»;
- математическая модель протокольного автомата сетевой программы и метод ее проецирования на бинарный код;
- автоматизированный метод решения задач построения протокольного автомата и его уточнения с использованием бинарного кода «горячей» трассы выполнения программы;
- макеты инструментов для восстановления форматов сообщений сетевых протоколов и построения протокольного автомата в ходе анализа трассы выполнения программы и фаззинга сетевых протоколов, чувствительных к сохранению состояния, в памяти тестируемого процесса, с использованием «плавающей» точки запуска порождающего сервера.

Проведены тесты на примере реальной сетевой программы и представлены полученные результаты. В дальнейшем работы по данной тематике будут продолжены в соответствии с определенными перспективными задачами и направлениями.

Список литературы / References

[1] SMACC – State Machine Asynchronous C++. Available at: https://smacc.dev/behavioral-vs-protocol-state-machines/#, accessed 11.10.2021.

[2] Poll E. LangSec meets state machines. Presentation at the IT day at SWIFT, Belgium, 2017. Available at: https://www.cs.ru.nl/E.Poll/talks/SWIFT_2017.pdf, accessed 11.10.2021.

[3] Székely G., Ládi G. et al. Protocol State Machine Reverse Engineering with a Teaching-Learning. Acta Cybernetica, vol. 25, issue 2, 2021, pp. 517-535.

[4] Ládi G., Buttyán L., Holczer T. GrAMeFFSI: Graph analysis based message format and field semantics inference for binary protocols using recorded network traffic. Infocommunications Journal, vol. 12, issue 2, 2020, pp. 25-33.

[5] Shahbaz M., Groz R. Inferring mealy machines. Lecture Notes in Computer Science, vol. 5850, 2009, pp. 207-222.

[6] Sija B.D, Goo Y.-H. et al. A Survey of Automatic Protocol Reverse Engineering Approaches, Methods, and Tools on the Inputs and Outputs View. Security and Communication Networks, 2018, Article ID 8370341, 17 p.

[7] Shevertalov M., Mancoridis S.. A reverse engineering tool for extracting protocols of networked applications. In Proc. of the 14th Working Conference on Reverse Engineering (WCRE ’07), 2007, pp. 229-238.

[8] Xiao M.-M., Yu S.-Z., and Wang Y. Automatic network protocol automaton extraction. In Proc. of the 3rd International Conference on Network and System Security (NSS ’09), 2009, pp. 336-343.

[9] Trifilo A., Burschka S., Biersack E. Traffic to protocol reverse engineering. In Proc. of the IEEE Symposium on Computational Intelligence for Security and Defense Applications, 2009, pp. 1-8.

[10] Antunes J., Neves N. Building an automaton towards reverse protocol engineering. 2009. Available at: http://www.di.fc.ul.pt/~nuno/PAPERS/INFORUM09.pdf, accessed 11.10.2021.

[11] Antunes J., Neves N., Verissimo P. Reverse engineering of protocols from network traces. In Proc. of the 18th Working Conference on Reverse Engineering (WCRE ’11), 2011, pp. 169-178.

[12] Wang Y., Zhang Z. et al. Inferring protocol state machine from network traces: a probabilistic approach. In Proc. of the 9th Applied Cryptography and Network Security International Conference (ACNS ’11), 2011, pp. 1-18.

[13] Zhang Z., Wen Q.-Y., and Tang W. Mining protocol state machines by interactive grammar inference. In Proc. of the 2012 3rd International Conference on Digital Manufacturing and Automation (ICDMA ’12), 2012, pp. 524-527.

[14] Laroche P., Burrows A., and Zincir-Heywood A.N. How far an evolutionary approach can go for protocol state analysis and discovery. In Proc. of the IEEE Congress on Evolutionary Computation (CEC ’13), 2013, pp. 3228-3235.

[15] Meng F., Liu Y. et al. Inferring protocol state machine for binary communication protocol. In Proc. of the IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA ’14), 2014, pp. 870-874.

[16] Borisov N., Brumley D.J. et al. Generic application-level protocol analyzer and its language. MSR Technical Report MSR-TR-2005-133, 2005, 15 p.

[17] Wang Y., Li X. et al. Biprominer: automatic mining of binary protocol features. In Proc. of the 12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT ’11), 2011, pp. 179-184.

[18] Bossert G., Guihéry F., and Hiet G. Towards automated protocol reverse engineering using semantic information. In Proc. of the 9th ACM Symposium on Information, Computer and Communications Security, 2014. Pp. 51-62.

[19] Bossert G., Guihéry F. Reverse and simulate your enemy botnet C&C. Mapping a P2P Botnet with Netzob. In Proc. of the Black Hat Abu Dhabi Conference, 2012, 21 p.

[20] Luo J.-Z. and Yu S.-Z. Position-based automatic reverse engineering of network protocols. Journal of Network and Computer Applications, vol. 36, issue 3, 2013, pp. 1070-1077.

[21] Comparetti P.M., Wondracek G. et al. Prospex: protocol specification extraction. In Proc. of the 30th IEEE Symposium on Security and Privacy, 2009, pp. 110-125.

[22] Xiao M.-M. and Luo Y.-P.. Automatic protocol reverse engineering using grammatical inference. IFS, vol. 32, no. 5, pp. 3585–3594, Apr. 2017, DOI: 10.3233/JIFS-169294.

[23] Goo Y.-H., Shim K.-S. et al. Protocol Specification Extraction Based on Contiguous Sequential Pattern Algorithm. IEEE Access, vol. 7, 2019, pp. 36057-36074.

[24] List of (automatic) protocol reverse engineering tools/methods/approaches for network protocols. Available at: https://github.com/techge/PRE-list, accessed 14.10.2021.
[25] Costa M., Crowcroft J. et al. Vigilante: End-To-End Containment of Internet Worms. In Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005), 2005, pp. 133-147.
[26] Lang K.J. Faster Algorithms for Finding Minimal Consistent DFAs. Technical Report. NEC Research Institute, 1999, 19 p.
[27] Needleman S. and Wunsch C. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. Journal of Molecular Biology, vol. 48, issue 3, 1970, pp. 443-453.
[28] Grammar-Mutator. Available at: https://github.com/AFLplusplus/Grammar-Mutator, accessed 22.09.2022.
[29] AFLplusplus. Available at: https://github.com/AFLplusplus/AFLplusplus.git, accessed 20.09.2022.

Информация об авторах / Information about authors

Иван Владимирович ШАРКОВ – научный сотрудник отдела компиляторных технологий. Сфера научных интересов: информационные технологии, системное программирование, фаззинг, компьютерные сети, исследования программ.

Ivan Vladimirovich SHARKOV is a researcher at the compiler technologies department. His research interests include information technologies, fuzzing, networks, software researching.