DOI: 10.15514/ISPRAS-2021-33(3)-6

HTTP-Request Classification in Automatic Web Application Crawling

A.V. Lapkina, ORCID: 0000-0002-7249-7672 <amiriya@seclab.cs.msu.ru>
A.A. Petukhov, ORCID: 0000-0002-1427-2440 <petand@seclab.cs.msu.su>
Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia

Abstract. The problem of automatic requests classification, as well as the problem of determining the routing rules for the requests on the server side, is directly connected with analysis of the user interface of dynamic web pages. This problem can be solved at the browser level, since it contains complete information about possible requests arising from interaction interaction between the user and the web application. In this paper, in order to extract the classification features, using data from the request execution context in the web client is suggested. A request context or a request trace is a collection of additional identification data that can be obtained by observing the web page JavaScript code execution or the user interface elements changes as a result of the interface elements activation. Such data, for example, include the position and the style of the element that caused the client request, the JavaScript function call stack, and the changes in the page's DOM tree after the request was initialized. In this study the implementation of the Chrome Developer Tools Protocol is used to solve the problem at the browser level and to automate the request trace selection.

Keywords: request classification; application crawling; dynamic web application; Chrome DevTools

For citation: Lapkina A.V.**,** Petukhov A.A. HTTP-Request Classification in Automatic Web Application Crawling. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 3, 2021, pp. 77-86. DOI: 10.15514/ISPRAS-2021-33(3)-6.

Классификация HTTP-запросов к серверу в задаче автоматического обхода современных веб-приложений

А.В. Лапкина, ORCID: 0000-0002-7249-7672 <amiriya@seclab.cs.msu.ru>
А.А. Петухов, ORCID: 0000-0002-1427-2440 <petand@seclab.cs.msu.su>
Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. Задача автоматической классификации запросов приложения, а также задача определения правил маршрутизации запросов на сервере напрямую связана с анализом пользовательского интерфейса динамических веб-страниц и может быть решена на уровне браузера, поскольку он содержит полную информацию о возможных запросах, возникающих при взаимодействии пользователя с каждой из страниц веб-приложения. В данной работе для решения поставленной задачи предлагается использовать данные из контекста выполнения запроса в веб-клиенте с целью выделения дополнительных признаков для классификации запросов. При этом в качестве контекста возникновения или трассы запроса рассматривается совокупность дополнительных идентификационных данных, которые можно получить, наблюдая за выполнением JavaScript-кода на веб-странице или за изменением элементов пользовательского интерфейса в результате активации интерфейсных элементов. К таким данным, например, можно отнести положение и стиль элемента, вызвавшего клиентский запрос, стек вызовов функций JavaScript и изменение в DOM-дереве страницы после запроса. В рамках данной работы для автоматизации выделения трасс запросов и их последующей классификации используется реализация протокола Chrome DevTools.

Ключевые слова: классификация запросов; динамические веб-приложения; автоматический обход приложений; протокол Chrome DevTools.

Для цитирования: Лапкина А.В., Петухов А.А. Классификация HTTP-запросов к серверу в задаче автоматического обхода современных веб-приложений. Труды ИСП РАН, том 33, вып. 3, 2021 г., стр. 77-86 (на английском языке). DOI: 10.15514/ISPRAS–2021–33(3)–6.

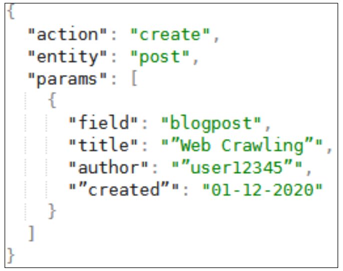
1. Introduction

The problem of classifying the requests from a web application client to a server and correlating them with application functions most often arises while analyzing applications using the black box method [1]. In the case of automated web application testing, the first step is collecting information about it. The structure of the application, its functions, input parameters, and types of requests are investigated. To collect this information, it is required to solve the problem of navigating the web application interface [2] – to find control elements automatically and activate them in order to cause client-server interaction.

To make sensible decisions in the navigation process, it is necessary to determine the results of triggering an action in the web interface: what HTTP request will be sent to the server, which function of the application will be executed, and how the state of the web application will change.

Since modern web interfaces are built with HTML and JavaScript technologies, the problem of navigating the application is reduced to analyzing the web interface (DOM and its visual presentation) and Javascript code. The latter implements the logic for the user and the server interaction: it processes user actions in the web interface, sends requests to the server and displays the results of their execution.

A particular problem in the process of navigating a web application is connected with correlating outgoing requests with the server-side actions of the web application. In traditional web applications, functions were uniquely addressed by URLs, so the problem of matching a request to an action on the server-side was trivial. In modern web applications, especially in single-page applications that implement the RPC concept (JSON RPC, XML RPC), URL can be the same for all server-side actions and the name of the function can be passed in the request parameters (see fig. 1). In order to correlate outgoing requests with the functions of the web application, it is necessary to extract a set of features from outgoing requests that uniquely identify functions of the web application.



```
{
  "action": "create",
  "entity": "post",
  "params": [
    {
      "field": "blogpost",
      "title": "”Web Crawling”",
      "author": "”user12345”",
      "”created”": "01-12-2020"
    }
  ]
}
```

Fig.1 Example of a POST-request with JSON in the body.
The called function is passed in the action field of the JSON structure.

Modern web applications use the concept of incoming requests routing [3]. To associate an incoming HTTP-request with a specific function or class in the application code, the developer defines the request routing rules: a table with predicates for HTTP-requests and function names. To process the next incoming request, the predicate for functions are calculated and the one that returns true will be called (the table is looked up from the top to the bottom until the first routing rule is triggered). The minimum set of request parameters, which values make the predicate true, will be called the discriminant for this request.

The set of specific values of the discriminant's parameters, that allow us to classify the request explicitly, is considered as the request key. In example presented on fig. 3 "*action*": "*create*" pair is the request key.

For requests with body-parameters in the JSON format, we will consider the ones with Content-Type: *application/x-www-form-urlencoded* and take into account not only the name of the significant parameters, but also the nesting objects degree.

In the paper, sites that use ReactJS library and implement a web interface in accordance with the framework rules specified in the documentation [4, 5] are investigated. This decision was made as ReactJS is one of the most popular framework among sites written with JavaScript.

2. Related work

The problem of classifying web application requests consists of two main subproblems. The first one is connected with a strategy for obtaining a set of outgoing requests of the web application. The second one is connected with determining a strategy for the inductive extraction of classification features.

The strategy of building a set of outgoing requests determines the order the application interfaces would be processed, and the order controls (links, buttons, tabs, scrolling, etc.) implemented in the graphical interface will be activated. The problem of automatic construction of the outgoing requests set can be solved with web crawlers using such methods as depth-first crawling, breadth-first crawling, or random crawling [6]. However, these strategies are ineffective for modern dynamic web applications [7, 8].

In modern surveys, the use of dynamic analysis of the web applications [8, 9], as well as additional properties of the web pages is used to solve this problem and to improve the quality of crawling. For example, they consider using the analysis of the structure of the web page elements and their relative position, as well as the history of elements crawling [10] or the user interface segmentation [11].

Traditionally, such request data elements as a method, target URL, path and GET- or POST-parameters are used as features for classifying outgoing requests. However, in order to facilitate the requests classification, some studies consider additional indicators related to the state of the web application at the moment the request was initialized. For example, the state of a hierarchical finite state machine built in the process of navigating the application [12] or the state of the DOM model of the page [13] is used as such additional features.

3. General design

In this research, the problem of constructing a classifier of outgoing HTTP requests from a web client to a web application, that allows us to restore the routing model on the server-side of the application as part of automatic website crawling is solved by developing the algorithm of classification. The classifier receives a site to crawl as an input. The result of the tool's operation is a set of discriminants. Their combined values are the key to identify the action on the server-side for each request.

Automatic forms filling [14] and navigating the internal zone of a web application are not considered in this paper. The lattest means that if the access to the internal zone of the web application requires authentication [15] is not considered in this paper.

It was assumed that the context of outgoing requests may contain parameters that can be used as identification keys of the actions on the server side. If such parameters are found, it is suggested to use them as additional features for identifying the requests. It was also assumed that it is possible to build an iterative algorithm for classifying outgoing requests based on the found key parameters from the context. Since the URL is provided as an input, elements are activated gradually and the set of requests is formed iteratively. That is the reason it was decided to select the request features gradually.

In the next sections a description of the approach, implementation and results of experiments evaluating the validity of the assumption and the applicability of the approach are situated.

The task of selecting additional features requires a preliminary analytical study of the relations between user actions and the parameters of the request context. The research is performed for applications built on the basis of the ReactJS library [16]. The unified concept of programs that use this framework allows extrapolating the results obtained on the experimental set of sites to other sites based on this technology.

To establish the dependency between the context and the outgoing requests parameters, it is necessary to mark up some data manually and analyze the frequency of occurrence of significant context parameters types. If it turns out that there are such sets of parameters in the context that will have the same set of values (key), when two identical actions from the web interface are triggered, and which values would be different, when different actions are triggered, then we assume that there is a dependency between context parameters and classes of outgoing application requests.

In this paper, such context elements as the DOM state before the request was sent, the DOM state after the request was sent, the identifier of the DOM element node to which the called event handler belongs, the style of this element and the call frames array (the stack trace or the list of called functions with script identifiers and function positions) are examined.

The preliminary experimental reseach consists of several steps. As a first step the same action A is triggered via two different interface elements on the selected site performing interactions A_1 and A_2. Their traces T_1 and T_2 with the sets of parameters DOM_before_1, DOM_after_1, $node\_id_1$, css_1, $callframes_1$ and DOM_before_2, DOM_after_2, $node\_id_2$, css_2, $callframes_2$ are obtained. Then action B with trace T_3, different from actions A is triggered. After that, the values of the traces T_1, T_2 and T_3 are compared. The next step is to determine which parameters from the traces T_1 and T_2 have coinciding values and which parameters in pairs T_1, T_3 and T_2, T_3 have different values. After that the same comparison is made for other actions on the selected site and on other sites from the sites list. If results of the experiment show that there is a set of context parameters where with a high probability the same values are used for the same actions and where different actions result in different values, then they will be used as additional classification features.

Site list for experimental research was obtained from the Built With list [17] and the top sites of Coder Academy [18]. To select significant parameters,sites with different user interface complexity were used: from very complex (airbnb.com, facebook.com) to simpler ones (bbc.com, bleacherreport.com). The list also included sites with different routing schemes, such as routing by URL, routing based on query-parameters or routing based on body-parameters of the POST-requests. These requirements were intended to provide better coverage of various site types used on the Internet.

The experiment of analyzing dependency between significant context parameters and user actions was carried out on 20 target sites. The results are presented in Table 1 and Table 2.

Table 1. Percentage of coincidence between actions and context parameters for identical actions

DOM before action	DOM after action	node id	css	callframes
58%	54%	80%	65%	96%

Table2. Percentage of difference between actions and context parameters for different actions

DOM before action	DOM after action	node id	css	callframes
81%	92%	99%	73%	100%

The experiment results show that the strongest dependency corresponds to the callframes parameter. In this regard, it was decided to use the callframes array from the request context to classify requests to the server in addition to such request's attributes as its method, URL, path, query-parameters and body-parameters for POST-requests.

To validate the suggested approach, a classification algorithm was composed and tested. It receives a site for processing as an input, and produces a set of request's discriminants as an output.

4. Classification Algorithm

The request classification algorithm implements the idea of inductive constructing a set of significant features. An example of the basic algorithm processing two user events A and B is presented below.

Data structures used:

VP ($valuable\ parameters$): a set of significant request parameters. Consists of elements in the form ($param\_name$: [$val1, val2, val3, \dots$]. Initially $VP = \emptyset$.

HP ($hint\ parameters$): a set of possibly significant parameters. $HP = \emptyset$.

NVP ($not\ valuable\ parameters$): a set of non-significant query parameters. Initially $NVP = \emptyset$.

AP ($all\ parameters$): set of all request parameters. Consists of elements in the form ($param\_name$: (val_1: $counter\_1$, val_2: $counter\_2$), where $param\_name$ is the name of the parameter, val_i is the i-th value of this parameter, $counter\_i$ is the number of times that the value of the $param\_name$ parameter has been encountered with the value val_i }. Initially $AP = \emptyset$.

RS ($request\ schemes$): A set of application request schemes. Each request scheme is a structure with fields containing the method, hostname, path, callframes, and the names of the get and post parameters. $RS = \emptyset$.

$trace, trace2$: the trace of the request. Consists of hostname, path, callframes, query-parameters (if any) and body-parameters (if any).

$P, P2$ ($parameters$): variable to store the parameters of the current request.

$counter$: requests counter. Initially $counter\ =\ 0$.

Used procedures:

$CheckScheme$ (S): Checks the presence of Scheme S in the RS set. Returns true if schema S is present in RS, false otherwise (see Algorithm 1).

```
Data: scheme S, set of all schemes RS
Result: boolean value that indicates if S is present in RS
for scheme in RS do
    if ((hostname in S = hostname in scheme) and
    (path in S = path in scheme) and (method in S = method in scheme)
    then
       return true;
    end
    if (callframes in S = callframes in scheme)
    then
       return true;
    end
    if ((query-params in S = query-params in scheme)
    and (body-params in S = body-params in scheme))
    then
       return true;
    end
    return false;
end
```

Algorithm 1: CheckScheme

Technical aspects such as extracting custom events from the web pages for crawling, navigating between application pages, and triggering custom event handlers, are discussed in the section Implementation.

The basic logic of the algorithm is presented in Algorithm 2.

```
Data: two custom event handlers A, B received from a given site
      for crawling
Result: a set of discriminants for custom events A, B
1 trigger event listener A;
2 intercept trace;
3 counter+ = 1;
4 VP ← hostname, path (where hostname, path ∈ trace);
5 P ← query-params; body-params (where query-params,
  body-params ∈ trace);
6 for param in P do
7    if ((param in AP) and (param.value = AP.param name.val_i))
     then
8      counter i+ = 1
9    else
10     AP ← {param:value : 1}
11   end
12   if param in NVP
     then
13     remove param from P;
14   end
15 end
16 AP ← P;
17 S ← hostname, path, callframes, query-params, body-params
18 (where hostname, path, callframes, query-params, body-params ∈ trace);
19 if checkScheme (S) = true
    then
20   trigger event listener B;
21   counter+ = 1;
22   repeat steps 4-41;
23   HP ← P;
24   trigger event listener A;
25   intercept trace2;
26 else
27 end
28 P2 ← query-params; body-params
   (where query-params, body-params ∈ trace2);
29 VP ← PP2;
30 NVP ← (PP2)/(PP2);
31 for param in NVP do
32   remove param from VP;
33   remove param from S;
34   for scheme in RS do
35      remove param from scheme;
36   end
37 end
38 RS ← S;
39 trigger event listener B;
40 counter+ = 1;
41 repeat steps 4 -41;
```

Algorithm 2. Basic classification algorithm

In a general case, the algorithm sequentially processes all activated user events for a given site. When the work is complete, the number of parameters and their values are recalculated from the set of all application parameters. In this case, the parameters that had the same value for all processed requests are moved from the list of significant parameters (if they were there) to the list of insignificant ones, and are also removed from the request schemes (see Algorithm 3).

```
Data: sets AP, VP, NVP, SR and counter variable
Result: a set VP for application requests
for param in AP do
    if (length(param) = 1) and (counter = param.counter)
    then
        remove param from VP
        remove param from SR
        NVP ← param
    end
end
```

Algorithm 3: Algorithm for recalculating the significance of parameters

The output of the algorithm is a set of significant request parameters. In this case, the key from the values of these discriminants allows the outgoing application request to be uniquely identified.

To validate that the constructed algorithm is applicable, a tool was developed that implements the suggested classifier. It iteratively constructs the set of outgoing requests for the application and extracts the classification features.

5. Implementation

The constructed tool automatically performs the following actions in the process of building a set of outgoing requests in automatic mode:

- collects custom event handlers used on the page;
- activates the handlers obtained in step 1, thus initiating the HTTP request from the client to the server;
- determines the content of emerging HTTP requests;
- defines the context of emerging requests;
- monitors dynamic changes in the DOM of a web page;
- extracts the discriminants of request taking into account the requests' context according to the basic algorithm.

From an architectural point of view, the classifier can be divided into the following logical components (see fig. 2).

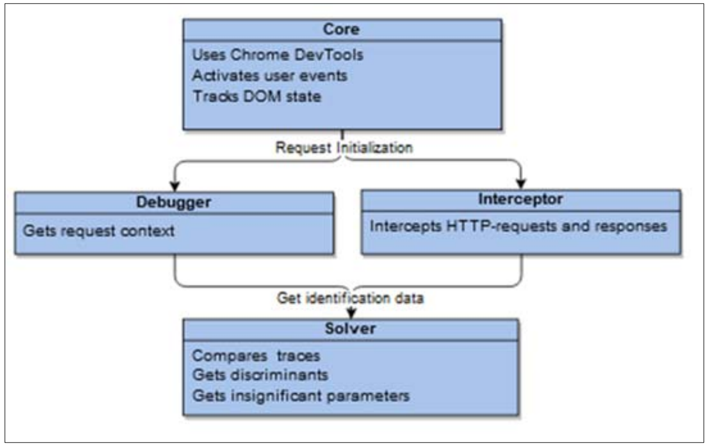


Fig. 2. Tool components

The core of the classifier is responsible for interacting with the browser and using the Chrome DevTools protocol. This protocol is a programmable version of the developer's toolkit for Chromium browsers. In the study it is used to navigate a web application by automatically activating user events on a web page, as well as to track the state of the browser context at the time when HTTP-requests are performed.

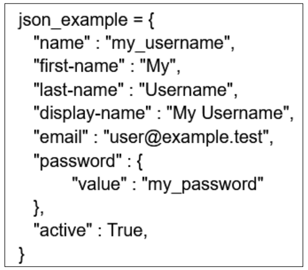
Debugger is used to get the context of the HTTP request and extract the callframes for further processing.

An interceptor is used to intercept requests from the client side of the application, as well as to obtain request elements such as URL, path, and parameters.

Solver represents the classifier itself. It compares the request traces received from the debugger and the interceptor. This part is also responsible for making decisions about the significance of the received features for the classification. It selects discriminants of requests and forms a list of parameters that are not significant for subsequent classification.

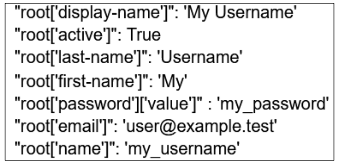
Possible complexity of the parameters' structure must be considered while examining request elements and their contexts.

For a more convenient representation of data transmitted in JSON format in the current study the DeepDiff library was used. It allows users to represent data as a set of fields and values, taking into account nested elements (see fig. 3 and fig. 4).



```
json_example = {
   "name" : "my_username",
   "first-name" : "My",
   "last-name" : "Username",
   "display-name" : "My Username",
   "email" : "user@example.test",
   "password" : {
          "value" : "my_password"
   },
   "active" : True,
}
```

Fig. 3. Data in JSON format



```
"root['display-name']": 'My Username'
"root['active']": True
"root['last-name']": 'Username'
"root['first-name']": 'My'
"root['password']['value']" : 'my_password'
"root['email']": 'user@example.test'
"root['name']": 'my_username'
```

Fig. 4. Same JSON data after DeepDiff processing

6. Experiments

The implemented classifier was firstly tested manually on 10 sites built with ReactJS. For this experiment the activation of user events was performed manually through interaction with the web interface of the application. The requests interception, their contexts selection and subsequent classification were performed automatically. The analysis of the discriminants extracted during the classification showed their 100\% completeness. In other words, there were no parameters that have been mistakenly marked as insignificant based on the classification results. The results of this experiment support the suggested method of solving the problem and allow proceeding to an automatic experiment.

To test the classifier in automatic mode, from the constructed set of 100 sites built with ReactJS, sites using Captcha were excluded. As a result, the final set consisted of 96 sites. The subsequent analysis of the received discriminants of requests also showed their completeness and confirmed the possibility of classifying the requests of the web application using their context. Moreover, usage of callframes helped to classify requests for 73\% of the sites crawled. Therefore, the experiment was considered as successful and the suggested approach was verified and showed its applicability in case of sites, written with React. Nevertheless, to expand the research results to the sites built with other frameworks, additional experiments are required.

In addition, due to the approach of activating custom events twice, using their context and removing insignificant request elements, it was possible to reduce the number of distinguished request discriminants for 52% in comparison with the total number of parameters received. This means that the number of parameters for fuzzing decreased and therefore the process of the subsequent black box testing may become more efficient.

This notwithstanding, in the left 48% of parameters that were marked as valuable, there may be some that were falsely recognized as significant. Nevertheless, the task of identifying was not considered in this study.

Based on the results of the experiments, the influence of request parts on routing was also calculated. Their frequency of occurrence is presented on fig. 5.

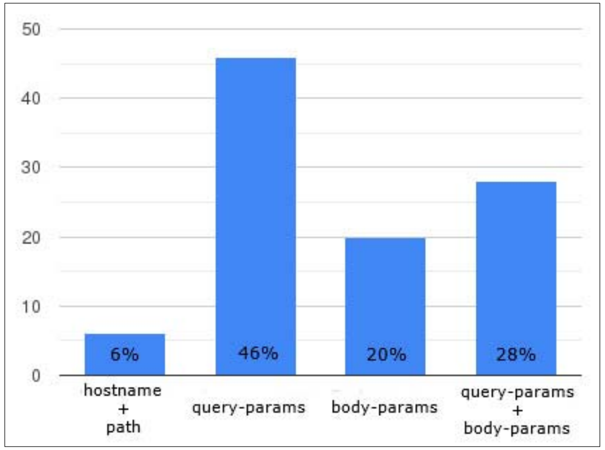


Fig. 5. Influence of request elements on routing in percents

7. Conclusion

The paper suggests a method for classifying requests of web applications with a dynamic interface. The experiments show that the suggested method, based on the usage of request context as a source for additional classification features solves the problem of classifying requests with the same level of completeness as the naive method that takes into account only the request content. The constructed classifier helps to reduce the number of insignificant parameters among the discriminants of the request, which is a positive achievement in the case of using a tool for determining the parameters of application requests for subsequent black box testing.

Список литературы / References

[1] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In Proc. of IEEE Symposium on Security and Privacy, 2010, pp. 332-345.

[2] A.M. Reina-Quintero. Surveying navigation modelling approaches. International Journal of Computer Applications in Technology, vol. 33, no. 4, 2008, pp. 327-336.

[3] P. Himschoot. Single Page Applications and Routing. In Blazor Revealed. Building Web Applications in .NET. Apress, 2019, pp. 187-212.

[4] ReactJS official web page. Available at http://www.ReactJs.org, accessed 10.03.2021.

[5] A. Fedosejev. React.js Essentials. A fast-paced guide to designing and building scalable and maintainable web apps with React.js. Packt Publishing, 2015, 208 p.

[6] C. Olston and M. Najork. Web crawling. Foundations and Trends in Information Retrieval, vol. 4, no. 3, 2010, pp. 175–246.

[7] S. Khalid, S. Khusro, and I. Ullah. Crawling ajax-based web applications: Evolution and state-of-the-art. Malaysian Journal of Computer-Science, vol. 31, no. 1, 2018, pp. 35–47.

[8] Г.М. Носеевич, А.А. Петухов. Поиск входных точек для веб-приложений с динамическим пользовательским интерфейсом. Безопасность информационных технологий, том 6, no.1, 2013 г., стр. 13-20 / G.M. Noseevich, A.A. Petuhov. Determining Data Entry Points for Javascript-rich Web applications. IT Security (Russia), vol. 6, no. 1, 2013, pp. 13-20 (in Russian).

[9] T. Pandikumar, Tseday Eshetu. Detecting Web Application Vulnerability using Dynamic Analysis with Penetration Testing. International Research Journal of Engineering and Technology, vol. 03, no. 10, 2016, pp. 430-433.

[10] А.А. Петухов, Н.Б. Матюнин. Автоматический обход веб-приложений с динамическим пользовательским интерфейсом. Проблемы информационной безопасности. Компьютерные системы, no. 3, 2014 г., стр. 43-49 / A.A. Petuhov, N.B. Matjunin. Automatic crawling of web applications with dynamic user interface. Information Security Problems. Computer Systems, no. 3, 2014, pp 43-49 (in Russian).

[11] И.С. Говорков. Оптимизация обхода страниц динамических веб-приложений, построенных с использованием библиотеки ReactJS. Тезисы конференции «Ломоносов-2018», 2018 г., стр. 1-3 / I.S. Govorkov. Optimization of crawling pages of dynamic web applications built using the ReactJS library. Abstracts of the conference «Lomonosov-2018», 2018, pp. 1-3 (in Russian).

[12] C. H. Liu, C. J. Wu, and H. M. Chen. Testing of AJAX-based Web applications using hierarchical state model. In Proc. of the IEEE 13th International Conference on e-Business Engineering (ICEBE), 2016, pp. 250-256.

[13] X. Zhang and H. Wang. AJAX Crawling Scheme Based on Document Object Model. In Proc. of the Fourth International Conference on Computational and Information Sciences (ICCIS), 2012, pp. 1198-1201.

[14] W.-K. Chen, C.-H. Liu, and K.-M. Chen. A web crawler supporting interactive and incremental user directives. Lecture Notes in Electrical Engineering, vol. 464, 2017, pp. 64–73

[15] H.Z.U. Khan. Comparative Study of Authentication Techniques. International Journal of Video Image Processing and Network Security, vol. 10, no. 04, 2010, pp. 9-13.

[16] S. Aggarwal. Modern Web-Development Using ReactJS. International Journal of Recent Research Aspects, vol. 5, no. 1, 2018, pp.133-137

[17] Websites using React. Available at https://trends.builtwith.com/websitelist/React, accessed 10.03.2021.

[18] Top 32 Sites Built with ReactJS. Available at https://medium.com/@coderacademy/32-sites-built-with-reactjs-172e3a4bed81, accessed 10.03.2021.

[19] Thends in JavaScript frameworks. Available at https://trends.google.com/trends/explore?q=vue.js,react,angular, accessed 10.03.2021

Информация об авторах / Information about authors

Анна Вадимовна ЛАПКИНА, магистр ВМК МГУ, выпуск 2021. Научные интересы: безопасность веб-приложений, обнаружение уязвимостей, автоматическая навигация по веб-приложениям.

Anna Vadimovna LAPKINA, master's graduate in 2021, CS department, MSU. Research interests: application security, vulnerability analysis, automatic application navigaton.

Андрей Александрович ПЕТУХОВ. Младший научный сотрудник в Лаборатории интеллектуальных систем кибербезопасности ВМК МГУ. Научные интересы: тестирование веб-приложений методом черного ящика, обнаружение уязвимостей, анализ программ.

Andrew Alexandrovitch PETUKHOV, researcher in Cybersecurity Lab, CS department, MSU. Research interests: black-box testing of web applications, vulnerability analysis, program analysis.