FARIS: Fast and Memory-efficient URL Filter by Domain Specific Machine

Yuuki Takano*, Ryosuke Miura*
*National Institute of Information and Communications Technology, Japan

E-mail: ytakano@wide.ad.jp, myu2@nict.go.jp

***Abstract*—Uniform resource locator (URL) filtering is a fundamental technology for intrusion detection, HTTP proxies, content distribution networks, content-centric networks, and many other application areas. Some applications adopt URL filtering to protect user privacy from malicious or insecure websites. AdBlock Plus is an example of a URL-filtering application, which filters sites that intend to steal sensitive information.**

Unfortunately, AdBlock Plus is implemented inefficiently, resulting in a slow application that consumes much memory. Although it provides a domain-specific language (DSL) to represent URLs, it internally uses regular expressions and does not take advantage of the benefits of the DSL. In addition, the number of filter rules become large, which makes matters worse.

In this paper, we propose the fast uniform resource identifier-specific filter, which is a domain-specific pseudo-machine for the DSL, to improve the performance of AdBlock Plus. Compared with a conventional implementation that internally adopts regular expressions, our proof-of-concept implementation is fast and small memory footprint.

***Keywords*-Network Security, URL Filter, Web**

I. Introduction

The Web has become a consequential platform for services on the Internet, and HTTP, which is the primary protocol of the Web, takes advantage of uniform resource locators (URLs) to identify locations or resources on the Internet. Therefore, URL filtering has also become a fundamental technology for services based on the Web.

URL filtering is adopted for several purposes. For example, it can be helpful for implementing parental controls, intrusion detection systems, content distribution networks, and content-centric networks. Notably, some HTTP proxy applications are capable of URL filtering. Both Privoxy [1] and SquidGuard [2], the latter being a URL redirector for the Squid HTTP proxy [3], have mechanisms for URL filtering.

AdBlock Plus [4] is one of the most popular browser extensions for filtering ad sites and other malicious sites. There are several types of filters that AdBlock Plus uses for ad and non-ad sites. For example, some filters are for Web tracking [5], [6], [7], which threaten the privacy of users, whereas some filters are for malicious or phishing sites. These filters are freely distributed on the Web (e.g., EasyList [8]), and users can protect themselves from such sites using these filters.

Because of the increase in online advertisements [9], the number of filter rules that AdBlock Plus utilizes has increased substantially. For example, EasyList [10], which is an official filter of AdBlock Plus, and a Japanese filter [11] have 45,929 and 9,600 rules, respectively. This implies that AdBlock Plus requires much memory and CPU resources to filter URLs.

AdBlock Plus provides a domain-specific language (DSL) [12] to describe URL-filtering rules and internally converts the language to regular expressions for URL filtering. Although the language is simpler than the corresponding regular expressions, AdBlock Plus does not take advantage of the language's simplicity. Therefore, in this paper, we propose a domain-specific pseudo-machine called the fast uniform resource identifier-specific filter (FARIS) to reduce memory and CPU resource consumption. The obtained efficiency should be useful not only for browsers but also for mobile devices, which have limited resources because of their size.

FARIS is based on a virtual machine (VM) approach for regular expressions [13], but for simplicity, it provides only four instructions. Moreover, to reduce memory consumption, it represents each instruction as a single byte. A conventional implementation adopts sequential search for matching URLs by multiple rules, which is inherently inefficient; however, FARIS uses hash tables for filter rule prefixes to increase matching throughput. Compared with a conventional implementation using regular expressions, our proof-of-concept implementation of FARIS was 23 times faster and consumed only 12.5% of memory. We distribute our implementation on our website [14] as open source software under BSD licensing for scientific reproducibility, and thus, anyone can use and modify it freely. Therefore, FARIS could be adopted for not only web browsers but also other mechanisms such as IDS, HTTP proxy.

II. Related Work

A. String and URL Pattern Matching

1) Exact Matching: The Knuth–Morris–Pratt algorithm (KMP) [15] is a fast and exact matching algorithm for strings. The Boyer–Moore algorithm [16] is also an algorithm for exact string matching and is faster than KMP. However, these algorithms cannot manage multiple query strings simultaneously.

The Aho–Corasick algorithm [17] is a well-known algorithm for multiple queries, but it explodes memory consumption when there are many queries. In [18], B.S. Michel et al. proposed a URL-hashing algorithm for Web caching. Their approach decomposes a URL into pieces and incrementally

TABLE I
FILTER SYNTAX OF ADBLOCK PLUS AND ITS REGULAR EXPRESSIONS

AdBlock's syntax	regular expression
*	`/.*/`
| of the beginning of a line	`/^/`
| of the end of a line	`/$/`
|| of the beginning of a line	`/[\w\-]+\/+/`
^	`/[\x00-\x24\x26-\x2C\x2F\x3A-\x40\x5B-\x5E\x60\x7B-\x7F]|$/`

applies a hash function to the pieces to minimize hash collisions. J.J. Garnica et al. [19] proposed an architecture for URL filtering in 100 GbE networks. Their architecture consists of two stages. The first stage consists of a filter made with a field programmable gate array; it filters packets based on a hash of the destination IP address. At the second stage, more complex software-based filtering is performed.

2) Prefix Matching: In addition to exact matching, some algorithms support prefix matching of URL filters. The PATRICIA trie [20] is a tree structure that represents multiple strings; a common prefix is internally represented by a shared node. Using this structure, algorithms can perform not only exact matching but also prefix matching. In [21], Z. Zhou et al. proposed a URL lookup engine. Their approach decomposes a URL into pieces and applies a hash function to each of the pieces. To enable prefix matching, they adopted a modified Wu–Manber algorithm [22]. In [23], N. Huang et al. proposed a hardware-based solution for the Wu–Manber algorithm. They made use of both binary content-addressable memory (CAM) [24] and ternary CAM in their architecture, and as a result, they achieved $O(1)$ complexity.

3) Regular Expressions: Regular expressions provide a more complex means of string pattern matching. It is well-known that a regular expression can be represented by both deterministic finite automaton and non-deterministic finite automaton (NFA). In addition, a regular expression can be represented by a specific VM [13] that simulates an NFA. In this study, we modified the VM approach for representing the DSL of AdBlock Plus to design a simple memory-efficient URL filter.

B. Applications of URL Filtering

AdBlock Plus [4] is one of the most popular URL-filtering software systems. It provides a DSL [12] for defining filter rules but does not take advantage of the benefits of the DSL. Instead, it internally uses regular expressions to represent filter rules. Accordingly, it suffers from much memory and CPU resource consumption [25], [26].

Privoxy [1] is an HTTP proxy and has a mechanism for filtering Websites based on regular expressions. SquidGuard [2], which is a URL redirector for the Squid HTTP proxy [3], also allows users to describe filter rules via regular expressions to filter URLs. These applications can take advantage of the filter rules of AdBlock Plus by translating the same rules into regular expressions; however, regular expression-based filtering consumes many computational resources, as noted above.

TABLE II
THE MACHINE INSTRUCTIONS OF FARIS

opcode	operand	operation
`char`	`c`	if `SP` is pointing to `c`, which is a character or a character set, increment `PC` and `SP`; otherwise, if the frame stack is empty, then abort matching, else pop `PC` and `SP` from the stack
`skip_to`	`c`	increment `SP` until it points to `c`; if `c` was not found, abort matching; otherwise, increment `PC` and push `PC + 1` and `SP` to the frame stack
`skip_scheme`		if `SP` is pointing to URL scheme, increment `SP` until it is not pointing to URL scheme; otherwise, abort matching
`match`		finish matching successfully

III. PRELIMINARIES

In this section, we describe the syntax and semantics of filter rules defined by AdBlock Plus [12]. Table I shows the syntax for pattern matching along with equivalent regular expressions. A rule for AdBlock Plus primarily consists of ASCII characters that are used to describe a URL [27] and special notation that includes `|` at the beginning or end of a line, `||` at the beginning of a line, and * and `^` to express a set of particular URLs.

This notation is used to efficiently express URLs. More specifically, * denotes a string of arbitrary length. `||` at the beginning of a line denotes a URL scheme (i.e., `http://`, `https://`, etc.), `^` denotes a separator, and so on. For example, filter rule `^example.com^8080^foo.php^u^url` will match `/example.com:8080/foo.php?u=url` as well as many other URLs.

Using the notation of AdBlock Plus's filter rules, URL filters can be efficiently and practically expressed. For example, `ads.com`, which is an exact pattern, does not distinguish between `http://ads.com/b.gif` and `http://ads.com/idx.html`; however, `ads.com^*.gif` will filter only the former.

IV. ARCHITECTURE OF FARIS

In this section, we present the architecture of FARIS, which is a domain-specific machine for AdBlock Plus's filter rules. FARIS can filter URLs efficiently because of its simple lightweight architecture.

A. Machine Instructions

FARIS is a bytecode interpreter. Thus, to perform pattern matching, AdBlock Plus's rules are translated into its machine instructions. FARIS interprets the four instructions

TABLE III
COMPILATION RULES FOR FARIS

input	instruction
⋆c	`skip_to` c
⋆^	`skip_to` *separator*
c	`char` c
^	`char` *separator*
|| of the beginning of a line	`char` *head* `skip_scheme`
| of the beginning of a line	`char` *head*
| of the end of a line	`char` *tail*

TABLE IV
BYTECODES OF FARIS

opcode	bytecode	priority
`skip_scheme`	`0x83`	high
`match`	`0x84`	high
`char`	`if (! 0x80 & code)`	low
`skip_to`	`if (0x80 & code)`	low

as follows: `char`, `skip_to`, `skip_scheme`, `match`. Furthermore, it has two registers, i.e., the string pointer (`SP`) and program counter (`PC`), as well as a frame stack for the `SP` and `PC`. In general, `char` reads a character, `skip_to` skips characters until a specified character is encountered, `skip_scheme` skips the URL scheme, and `match` indicates that the input string was matched successfully. Table II shows the instructions, operands, and what each instruction does (i.e., its semantics).

Table III shows compilation rules, with *separator*, *head*, and *tail* representing special characters to denote a separator, the beginning of a line, and the end of a line, respectively. Five translation rules that are applied before compiling filter rules are as follows: (1) if there are invalid characters in the input string, the string must be encoded by percent-encoding [27] to distinguish between reserved characters or words and others; (2) repeated ⋆ characters must be converted into a single ⋆ character; (3) ⋆ at the end of a line must be removed; (4) ⋆| at the end of a line must be removed; and (5) ^| at the end of a line must be replaced by ^ because ^ is a character set that includes a string terminator, i.e., if ^ is interpreted as a terminator and accepted, the following | is not accepted.

In addition, ⋆ and |⋆ at the beginning of a line, which are meaningless, must be removed to achieve further optimization, which is described in Section V-A.

B. Assembly Code and Bytecode

After translating a rule to an assembly code, the code is compiled into a bytecode, which is directly interpreted by FARIS. For memory efficiency, every instruction is encoded as a single byte; this means that both an opcode and its operand are encoded together in this single byte.

Table IV shows the valid bytecodes of FARIS. As shown in the table, `skip_scheme` and `match` are represented by `0x83` and `0x84`, respectively. Furthermore, `char` and `skip_to` are identified by the most significant bit (MSB); this means that a byte is interpreted as `char` if the code's MSB is zero; otherwise, the byte is interpreted as `skip_to`. Here, operands are encoded by bitwise OR operations with the given code. For example, `char c` and `skip_to c` are encoded as `(0x00 | c)` and `(0x80 | c)`, respectively, where | is the bitwise OR operator.

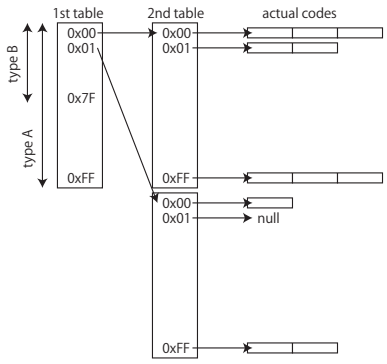


Fig. 1. Prefix Hash Table

To distinguish `skip_to` from `skip_scheme` and `match`, there are priorities between the instructions. When interpreting a bytecode, first, FARIS checks if the bytecode is `skip_scheme` or `match`; if the bytecode is not `skip_scheme` or `match`, it then checks if the bytecode is `char` or `skip_to`.

Note that *head*, *tail*, and *separator* are expressed as `0x7D`, `0x7E`, and `0x7F`, respectively. Table III shows these special characters.

V. OPTIMIZATION AND IMPLEMENTATION

In this section, we describe optimization techniques to increase processing speed, as well as how we implemented FARIS.

A. Prefix Hash Table

We adopted a prefix aggregation technique similar to that of the PATRICIA trie or Aho–Corasick algorithm. To avoid memory usage explosion, we introduced a prefix hash table indexed by prefixes of VM codes. Figure 1 shows how to construct the prefix hash table, which consists of first and second tables to indicate prefixes of codes. The second table points to an array of pointers for actual codes, each of which has the same prefix.

For matching, the instructions of the first table are used to match with a given query string, and if matched, the instructions of the second table are similarly used. Finally, the codes pointed to by the entry in the second table are used to accomplish matching.

We prepared two types of prefix hash tables and an array for codes that are not stored in the tables. The first table, denoted as type A, is for rules whose first and second codes are `char` *head*, and `skip_scheme`. Here, the third and fourth instructions are used as the indices of the first and second tables. The second table, denoted as type B, is for rules whose first instruction is not `char` *head*. Here, the first and second instructions are used as the indices. Other rules that do not fulfill the above conditions are stored in the array.

TABLE V
PARTICIPANTS OF WIDE CAMP 2015 SPRING

	Japanese		non-Japanese	
	student	non-student	non-student	total
male	26	67	9	102
female	5	1	0	6
total	31	68	9	108

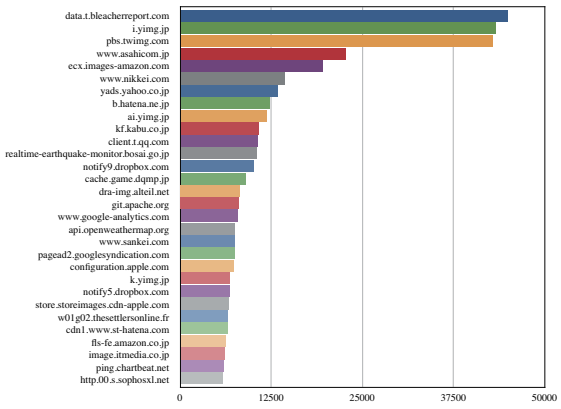


Fig. 2. Histogram of Top 30 Requested Domains

Note that the first table of type B needs only a range from `0x00` to `0x7F` because $\star$ at the beginning of a line is previously eliminated, as described in Section IV-A. As a result, the first table can be looked up by the most significant character of the input string with $O(1)$ complexity; this means that not all indices of the tables are required for matching. For example, if the most significant character of the input string is "a," `table["a"]` and `table[0x7F-0xFF]` should be selected for matching.

B. Multithreading

Simultaneous matching increases the performance of multicore CPUs. Therefore, we implemented a C++ class for multiple readers and single writer lock mechanism that uses atomic registers and applied this class to FARIS. Accordingly, multiple threads can simultaneously perform matching. Note that a single writer thread exclusively blocks all other threads, but it is practically negligible because filter updates are not performed very frequently, typically once per day.

C. Implementation

We implemented FARIS in C++0x11; We distributed our proof-of-concept implementation on the Web [14] as open source software under BSD licensing for scientific reproducibility; thus, anyone can use and modify it freely.

VI. EVALUATION

In this section, we detail the performance evaluations of our implementations using real HTTP requests.

A. Dataset

For our evaluation, we captured real HTTP requests at the WIDE Camp 2015 Spring [35], which is a workshop held from March 12–15, 2015 for researchers and operators of network technologies. The camp allowed researchers to conduct network experiments for research, and thus, we captured attendees' network traffic by mutual consent.

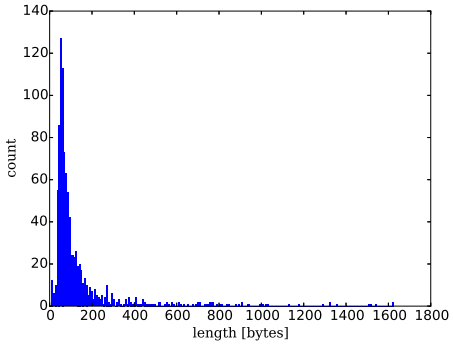


Fig. 3. Histogram of URL Length (1,000 Samples)

Table V summarizes the characteristics of the participants of the camp. There were 108 participants for four days, and almost all participants were Japanese. All student majors were network or related technologies, and all non-students were specialists in network technologies, including researchers, operators, and developers.

Overall, we captured 1,760,898 URL requests. Figure 2 shows the top 30 most requested domains. There were several types of domains, including news sites such as `www.nikkei.com` and `www.sankei.com`, social sites such as `bps.twimg.com` and `b.hatena.ne.jp`, and ad sites such as `yads.yahoo.co.jp` and `pagead2.googlesyndication.com`. For performance evaluation, we randomly selected 1,000 URLs from the captured data. Figure 3 shows a histogram of the length of the selected URLs. In this case, 50% and 95% of URLs are shorter than 76 and 561 bytes, respectively.

Table VI shows the filters we used for our experiments as well as statistics regarding the filters. There are two types of filters: URL and element filters. URL filters are used to filter URLs, whereas element filters are used to filter HTML elements, such as and . In this study, we focus entirely on URL filtering. Table VI shows that there were 77,139 URL filters in total, and they consisted of 1,529,108 `char`, 6,671 `skip_to`, and 59,131 `skip_scheme` instructions for FARIS.

For our evaluation, we defined three sets of filters because measurement results should be different for different datasets. These sets are as follows: (1) an EasyList set consisting of only EasyList [10]; (2) a set of typical filters consisting of EasyList, EasyPrivacy [28], and malware domains [29]; and (3) a set of all filters consisting of every filter shown in Table VI.

B. Targets and Measurement Environment

There were three types of implementations for evaluation: (1) an implementation without any optimization, as described in Section V; (2) an implementation with the prefix hash

TABLE VI
FILTER STATISTICS

filter	filter type		instructions		
	#url	#element	#char	#skip_to	#skip_scheme
easylist [10]	20,599	25,330	413,914	2,530	13,440
easyprivacy [28]	9,810	0	203,906	1,068	7,490
malware domains [29]	22,845	0	452,283	0	22,845
fanboy annoyance [30]	3,941	12,532	68,083	352	979
japanese [11]	9,600	0	127,302	481	5,063
japanese (tofu) [31]	1,511	352	24,471	84	1,102
easylist france [32]	2,856	1,645	72,424	652	2,548
easylist germany [33]	4,320	3,645	119,289	1,066	4,068
easylist italy [34]	1,657	1,221	47,436	438	1,596
total	77,139	44,725	1,529,108	6,671	59,131

TABLE VII
MEASUREMENT ENVIRONMENT

	MacBook Pro
CPU	Core i7 I7-4870HQ 2.5 [GHz], 4-core, HT turbo boost 3.7 [GHz]
OS	MacOS 10.10.2
C++ compiler	llvm clang++ 6.0

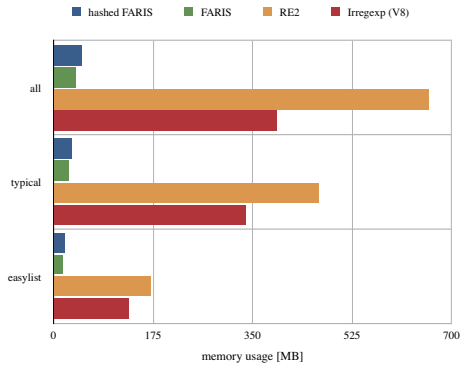


Fig. 4. Memory Usage

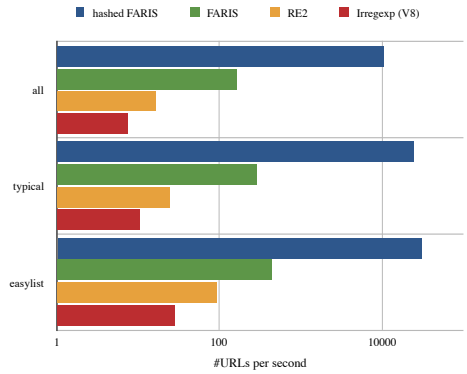


Fig. 5. Matching Throughput

table described in Section V-A; and (3) a multithreaded implementation with a prefix hash table.

Table VII shows the measurement environments. As shown in the table, we conducted experiments on a MacBook Pro.

For comparison, we evaluated the performance of Irregexp [36] via V8 [37] and RE2 [38], which are regular expression engines. More specifically, we implemented filters of AdBlock Plus's filter rules in JavaScript and C++ using RE2.

C. Performance Evaluation

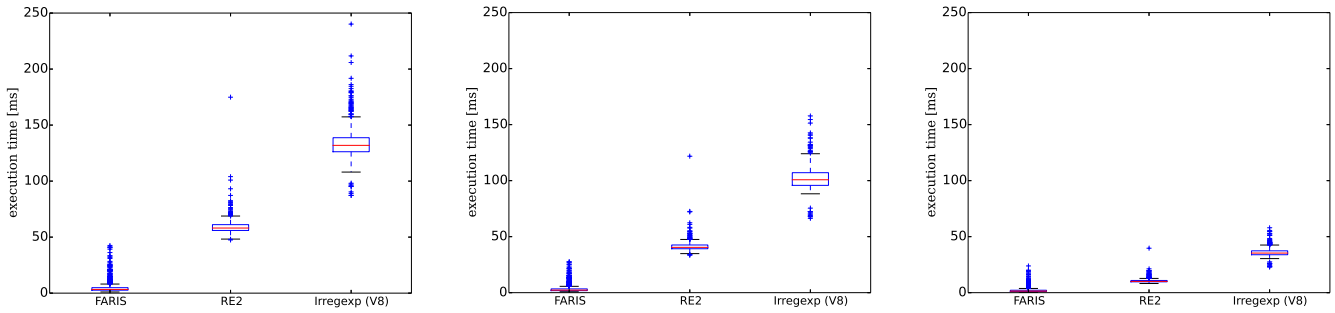
1) Memory Usage: Figure 4 shows results of our memory usage testing for FARIS and the regular expression engines. When using all filters, RE2 and Irregexp consumed approximately 660 and 390 MB of memory, respectively, but FARIS consumed only 38.1 MB memory. The figure reveals that FARIS requires only 6%–10% of the memory of the given regular expression engines.

Hashed FARIS requires additional memory for the hash table, but the increase is slight and negligible, consuming only 12.5% of memory as compared to the code written in JavaScript with all filters.

2) Throughput and Execution Time: Figure 5 shows results of our throughput testing for FARIS and the regular expression engines. When using all filters, FARIS could manage 162 URLs per second, respectively, but RE2 and Irregexp managed only 17 and 7 URLs per second, respectively. These results show that FARIS was 23 times faster than conventional implementations written in JavaScript. Hashed FARIS far outperformed the others, managing 10,676 URLs per second, which was 1,423 times faster than the sequeantil matching written in JavaScript.

Figure 6 shows the distributions of execution times for FARIS and other conventional implementations. When using all and typical filters, the execution times of RE2 and Irregexp were widely distributed as compared with FARIS, but when using EasyList, the distributions for RE2 and Irregexp became small, similar to FARIS. Figure 7 shows the distribution of the execution times of hashed FARIS, in which almost all queries were processed in 0.2 ms with every filter set.

3) Multicore Scalability: Figure 8 shows the throughput versus the number of worker threads on the CPU. With eight threads, hashed FARIS could manage 49,970 URLs per second. Because of our reader and writer lock implementations, the throughput linearly increased up to eight threads, at which the number of logical CPUs was equal to the number of threads. Note that the slope changed at four threads because of Intel's hyper-threading technology [39]. We confirmed



(a) all (b) typical (c) easylist

Fig. 6. Execution Times for Matching URLs

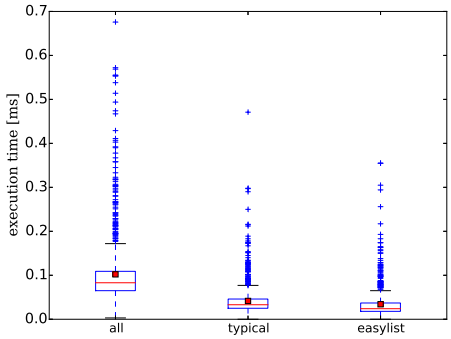


Fig. 7. Execution Times of Hashed FARIS for Matching URLs

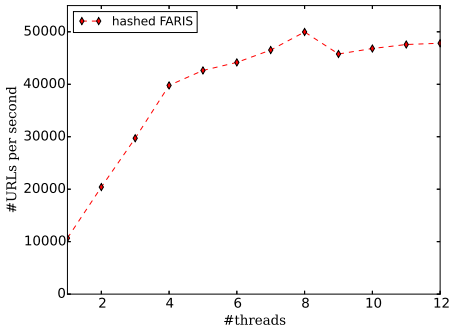


Fig. 8. Matching Throughput versus the Number of Threads (All Filters)

that using more than eight threads did not contribute to the throughput but rather suffered from the overhead of thread scheduling and context switches.

D. Theoretical Analysis

FARIS performs matching with $O(L \times M)$ on average and $O(L \times N \times M)$ in the worst case, where L, N, and M are the number of filter rules, average length of filter rules, and length of the input string, respectively. Hashed FARIS performs matching with $O\left(\frac{L \times M}{T} + \alpha\right)$ on average, where T and α are constant values depending on the size of the prefix hash table; this means that if the size of the table is sufficiently large and the prefixes of filter rules are uniformly distributed, the complexity of hashed FARIS will ideally approach $O(M)$ on average; however, the complexity of hashed FARIS equals that of FARIS in some outlier unusual cases and in the worst case. For example, if every filter rule has the same prefix, the complexities become equal to one another.

VII. Discussion

In this section, we discuss the applications or devices that we feel should use FARIS. FARIS should be quite suitable for Web browsers or browser extensions. AdBlock Plus is one of the most popular browser extensions, but it is implemented inefficiently. Using FARIS could increase AdBlock Plus's performance and reduce its large memory utilization. Thus, embedding FARIS into Web browsers or JavaScript engines is a good choice for improving overall performance.

Mobile devices have relatively limited computational resources and restricted power supplies in comparison with desktop or laptop PCs. Therefore, it is very important that fast and memory-efficient URL filtering is available to protect users from malicious sites. FARIS can also be helpful in protecting mobile devices because of its efficiency. In addition, it has the advantage that it can use AdBlock Plus's filter rules, which are freely available.

More specifically, regarding mobile devices, people tend to store their personal information on their mobile devices. Thus, malicious applications or Websites can steal such information for business purposes [40]. Although user tracking and targeted advertisements are becoming the essence of free online services [9], they should definitely cause privacy concerns [5]. Phishing targeting on mobile devices [41] is also a significant threat. URL filtering should help to protect mobile device users from these threats.

VIII. Conclusion

In this paper, we proposed FARIS, a pseudo-machine for a DSL provided by AdBlock Plus that implements fast and memory-efficient URL filtering. FARIS is based on a VM approach of regular expressions but provides only four instructions and represents each instruction as a single byte to reduce memory and CPU resource consumption.

Further, with FARIS, we proposed three optimization techniques. First, we proposed the prefix hash table as a method to avoid the sequential search that conventional implementations have made use of. Using this approach, we provide hash tables for prefixes of FARIS's instructions and use the tables to screen out filter rules when matching. Second, given that multithreading is a fundamental technology for maximizing CPU utilization, we implemented multiple reader and writer locks using an atomic register and applied this technique to FARIS.

For our experimental evaluations, we captured real HTTP requests from WIDE Camp 2015 Spring, acquiring 1,760,898 URLs in total, of which we used 1,000 randomly selected URLs. As a result, we showed that FARIS were 23 times faster and consumed only 10% and 12.5% of memory, respectively, in comparison with a conventional implementation written in JavaScript. Furthermore, we also showed that hashed FARIS was 1,423 times faster than sequeantial matching. We also revealed that the throughput of multithreaded FARIS is scalable on multiple CPU cores.

REFERENCES

[1] *Privoxy* (2016, June, 26) [Online], Available: http://www.privoxy.org/.

[2] *SquidGuard* (2016, June, 26) [Online], Available: http://www.squidguard.org/.

[3] *Squid* (2016, June, 26) [Online], Available: http://www.squid-cache.org/.

[4] *AdBlock Plus* (2016, June, 26) [Online], Available: https://getadblock.com/https://adblockplus.org/.

[5] J. R. Mayer and J. C. Mitchell, "Third-Party Web Tracking: Policy and Technology," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pp. 413–427, IEEE Computer Society, 2012.

[6] F. Roesner *et al.*, "Detecting and Defending Against Third-Party Tracking on the Web," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012* (S. D. Gribble and D. Katabi, eds.), pp. 155–168, USENIX Association, 2012.

[7] Y. Takano *et al.*, "MindYourPrivacy: Design and implementation of a visualization system for third-party Web tracking," in *2014 Twelfth Annual International Conference on Privacy, Security and Trust, Toronto, ON, Canada, July 23-24, 2014* (A. Miri, U. Hengartner, N. Huang, A. Jøsang, and J. García-Alfaro, eds.), pp. 48–56, IEEE, 2014.

[8] *EasyList* (2016, June, 26) [Online], Available: https://easylist.adblockplus.org/en/.

[9] P. Gill *et al.*, "Best paper - Follow the money: understanding economics of online aggregation and advertising," in *Proceedings of the 2013 Internet Measurement Conference, IMC 2013, Barcelona, Spain, October 23-25, 2013* (K. Papagiannaki, P. K. Gummadi, and C. Partridge, eds.), pp. 141–148, ACM, 2013.

[10] *EasyList (AdBlock Plus Filter)* (2016, June, 26) [Online], Available: https://easylist-downloads.adblockplus.org/easylist.txt.

[11] *Japanese Filter (AdBlock Plus Filter)* (2016, June, 26) [Online], Available: https://raw.githubusercontent.com/k2jp/abp-japanese-filters/master/abp_jp.txt.

[12] *Writing AdBlock Plus Filters* (2016, June, 26) [Online], Available: https://adblockplus.org/en/filters.

[13] R. Cox. *Regular Expression Matching: the Virtual Machine Approach* (2007, January) [Online], Available: https://swtch.com/~rsc/regexp/regexp2.html.

[14] Y. Takano. *FARIS: Fast and Memory-efficient URL Filter* (2016, June, 26) [Online], Available: https://github.com/ytakano/farisvm.

[15] D. E. Knuth *et al.*, "Fast pattern matching in strings," *SIAM Journal of Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[16] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 62–72, 1977.

[17] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[18] B. S. Michel *et al.*, "URL Forwarding and Compression in Adaptive Web Caching," in *Proceedings IEEE INFOCOM 2000, The Conference on Computer Communications, Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Reaching the Promised Land of Communications, Tel Aviv, Israel, March 26-30, 2000*, pp. 670–678, IEEE, 2000.

[19] J. J. Garnica *et al.*, "A FPGA-based scalable architecture for URL legal filtering in 100GbE networks," in *2012 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2012, Cancun, Mexico, December 5-7, 2012*, pp. 1–6, IEEE, 2012.

[20] D. R. Morrison, "PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric," *J. ACM*, vol. 15, no. 4, pp. 514–534, 1968.

[21] Z. Zhou *et al.*, "A High-Performance URL Lookup Engine for URL Filtering Systems," in *Proceedings of IEEE International Conference on Communications, ICC 2010, Cape Town, South Africa, 23-27 May 2010*, pp. 1–5, IEEE, 2010.

[22] S. Wu and U. Manber, "A Fast Algorithm For Multi-Pattern Searching," 1994.

[23] N. Huang *et al.*, "A Fast URL Lookup Engine for Content-Aware Multi-Gigabit Switches," in *19th International Conference on Advanced Information Networking and Applications (AINA 2005), 28-30 March 2005, Taipei, Taiwan*, pp. 641–646, IEEE Computer Society, 2005.

[24] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, vol. 41, no. 3, pp. 712–727, 2006.

[25] *AdBlock Plus's effect on Firefox's memory usage* (2014, May, 14) [Online], Available: http://www.reddit.com/r/programming/comments/25j41u/adblock_pluss_effect_on_firefoxs_memory_usage/.

[26] *On the Adblock Plus memory consumption* (2014, May, 14) [Online], Available: https://adblockplus.org/blog/on-the-adblock-plus-memory-consumption.

[27] T. Berners-Lee *et al.*, "Uniform Resource Identifier (URI): Generic Syntax." RFC 3986 (INTERNET STANDARD), Jan. 2005. Updated by RFCs 6874, 7320.

[28] *EasyPrivacy (AdBlock Plus Filter)* (2016, June, 26) [Online], Available: https://easylist-downloads.adblockplus.org/easyprivacy.txt.

[29] *Malware Domains (AdBlock Plus Filter)* (2016, June, 26) [Online], Available: https://easylist-downloads.adblockplus.org/malwaredomains_full.txt.

[30] *Funboy Annoyance (AdBlock Plus Filter)* (2016, June, 26) [Online], Available: https://easylist-downloads.adblockplus.org/fanboy-annoyance.txt.

[31] *Japanese Tofu Filter (AdBlock Plus Filter)*, (2016, June, 26) [Online], Available: http://tofukko.r.ribbon.to/Adblock_Plus_list.txt.

[32] *EasyList France (AdBlock Plus Filter)* (2016, June, 26) [Online], Available: https://easylist-downloads.adblockplus.org/liste_fr.txt.

[33] *EasyList Germany (AdBlock Plus Filter)* (2016, June, 26) [Online], Available: https://easylist-downloads.adblockplus.org/easylistgermany.txt.

[34] *EasyList Italy (AdBlock Plus Filter)* (2016, June, 26) [Online], Available: https://easylist-downloads.adblockplus.org/easylistitaly.txt.

[35] *WIDE Project* (2016, June, 26) [Online], Available: http://www.wide.ad.jp/.

[36] *Irregexp, Google Chrome's New Regexp Implementation* (2009, February, 4) [Online], Available: http://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html.

[37] *V8 JavaScript Engine* (2016, June, 26) [Online], Available: https://chromium.googlesource.com/v8/v8.git.

[38] R. Cox. *RE2* (2016, June, 26) [Online], Available: https://github.com/google/re2.

[39] *Intel®Hyper-Threading Technology* (2016, June, 26) [Online], Available: http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

[40] A. P. Felt *et al.*, "A survey of mobile malware in the wild," in *SPSM'11, Proceedings of the 1st ACM Workshop Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2011, October 17, 2011, Chicago, IL, USA* (X. Jiang, A. Bhattacharya, P. Dasgupta, and W. Enck, eds.), pp. 3–14, ACM, 2011.

[41] A. P. Felt and D. Wagner, "Phishing on mobile devices," in *In W2SP*, 2011.