FARIS: Fast and Memory-Efficient URL Filter on CPU and GPGPU

Yuuki Takano and Ryosuke Miura

National Institute of Information and Communications Technology, Japan
E-mail: ytakano@wide.ad.jp; myu2@nict.go.jp

Received 17 October 2016; Accepted 3 January 2017;
Publication 11 February 2017

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. Some web browser extensions, such as AdBlock Plus, provide a URL-filtering mechanism for sites that intend to steal sensitive information.

Unfortunately, these extensions are 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 ofthe benefits ofthe 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 dramatically improve the performance of some browser extensions. Compared with a conventional implementation that internally adopts regular expressions, our proof-of-concept implementation is fast and small memory footprint.

Keywords: URL filter, Web, online advertisement.

Journal of Software Networking, 101–136.
doi: 10.13052/jsn2445-9739.2017.006

1 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.

Some web browser extensions, such as AdBlock Plus [4] and uBlock [5], also provide a URL filtering mechanism, for filtering ad sites and other malicious sites. There are several types of filters that these extensions use for ad and non-ad sites. For example, some filters are for Web tracking [6–8], 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 [9]), and users can protect themselves from such sites using these filters.

Because of the increase in online advertisements [10], the number of filter rules that these extensions utilize has increased substantially. For example, EasyList [11], which is an official filter of AdBlock Plus, and a Japanese filter [12] have 45,929 and 9,600 rules, respectively. This implies that these extensions require much memory and CPU resources to filter URLs, and thus, the use of these extenions can cause serious performance penalties.

AdBlock Plus provides a domain-specific language (DSL) [13] to describe URL-filtering rules and internally converts the language to regular expressions for URL filtering. Other extensions, such as uBlock, also support the DSL and make use of regular expressions, too. Although the language is simpler than the corresponding regular expressions, These extensions do 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 [14], 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 [15] 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.

The main contributions of our work are:

- We have designed, implemented, and evaluated FARIS, afast and memory-efficient URL filter. Furthermore, our proof-of-cocnept implementation have been distrubited on the website.
- FARIS adopts AdBlock Plus's filter rules, which is one of the most popular and widely used for Web browser's URL filtering.
- We present a tiny pseudo machine architecture, which consists of 2 registers, 4 instructions, and 1 frame stack, to reduce memory and CPU utilization.
- We show that FARIS can be executed on both CPU and GPU.

Related work of URL filtering and string matching is described in Section 2. The architecture of FARIS is shown in Section 3. Optimization techniques for FARIS and implelentation are described in Section 4. Performance evaluation is shown in Section 5. Section 6 discussed pros and cons of FARIS, and it's applications. Finally, Section 7 concludes and gives some remarks on our proposal.

2 Related Work

2.1 String and URL Pattern Matching

2.1.1 Exact matching

The Knuth–Morris–Pratt algorithm (KMP) [16] is a fast and exact matching algorithm for strings. The Boyer–Moore algorithm [17] 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 [18] is a well-known algorithm for multiple queries, but it explodes memory consumption when there are many queries.

In [19], B. S. Michel et al. proposed aURL-hashing algorithm for Web caching. Their approach decomposes a URL into pieces and incrementally applies a hash function to the pieces to minimize hash collisions. J.J. Garnica et al. [20] 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.

2.1.2 Prefix matching

In addition to exact matching, some algorithms support prefix matching of URL filters. The PATRICIA trie [21] 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 [22], 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 [23]. In [24], N. Huang et al. proposed a hardware-based solution for the Wu–Manber algorithm. They made use of both binary content-addressable memory (CAM) [25] and ternary CAM in their architecture, and as a result, they achieved $O(1)$ complexity.

2.1.3 Hash-based matching

Hash-based algorithms are also used for URL filtering. The multi-level counting bloom filter (MLCBF) [26] is a URL filter algorithm based on the counting bloom filter (CBF) [27], a hashing structure. Compared to CBF and DLCBF [28, 29], MLCBF achieved reducing about 90 or 80% of memory utilization.

Yuan et al. propsed a hash-based URL matching algorithm named TFD [30]. To achieve fast and memory efficient URL matching, the TFD takes advantages of two-phase hash, finite state machine, and double-array storage [31] techniques. From their report, the TFD achieved 100 Mbps URL matching throughput over an x86 machine.

2.1.4 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 [14] 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. Some studies could achieve boosted regular expression performance using general-purpose computing on graphics processing units (GPG-PUs) [32–36]. Thus, in this study, we also attempt to take advantage of GPGPUs for our method.

2.2 Applications of URL Filtering

AdBlock Plus [4] and uBlock [5] are popular URL-filtering software systems. These systems provide a DSL [13] 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.

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.

Chou et al. proposed a content-based filter system on embedded Linux hoome gateway [37]. They take advantage of a URL filtering system for content-based filtering. SOBA [38] also provides a URL filtering mechanism for protecting teenagers from harmful information on online cyberspace. Unfortunately, URL filtering is also used for censorship [39].

Classifying URLs is also important to protect users from malicious Websites. Ma et al. describes several approaches for classifying suspicious URLs by using machine learning techniques [40–42]. Su et al. proposed a filtering mechanism based on multi-view analysis [43].

2.3 AdBlock Plus's Notations and Implementation

In this section, we describe the syntax and semantics of filter rules defined by AdBlock Plus [13]. Table 1 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 [44] 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.

Table 1 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	/$/
|| of the beginning of a line	/[\w\–]+\/+/
^	/[\x00-\x24\x26-\x2C\x2F\x3A-\x48\x5B-\x5E\x68\x7B-\x7F]I$/

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.

Figure 1 is actual JavaScript code used by AdBlock Plus to translate its rules into regular expressions. Although AdBlock Plus provides special notation, it uses regular expressions internally. In the given code, for efficiency, multiple * characters are removed from the beginning and end of a line.

```
1  // Remove multiple wildcards
2  var source = this.regexpSource
3    .replace(/\*+/g, "*")        // remove multiple wildcards
4    .replace(/\^\|$/, "^")       // remove anchors following separator
   placeholder
5    .replace(/\W/g, "\\$&")      // escape special symbols
6    .replace(/\\\*/g, ".*")      // replace wildcards by .*
7    // process separator placeholders (all ANSI characters but alphanumeric
   characters and _%.-)
8    .replace(/\\\^/g, "(?:[\\x00-\\x24\\x26-\\x2C\\x2F\\x3A-\\x40\\x5B-\\x5E\\
   x60\\x7B-\\x7F]|$)")
9    .replace(/^\\\|\\\|/, "^[\\w\\-]+:\\/+(?!\\/)(?:[^\\/]+\\.)?") // process
   extended anchor at expression start
10   .replace(/^\\\|/, "^")       // process anchor at expression start
11   .replace(/\\\|$/, "$")       // process anchor at expression end
12   .replace(/^(\.\*)/, "")      // remove leading wildcards
13   .replace(/(\.\*)$/, "");     // remove trailing wildcards
14
15 var regexp = new RegExp(source, this.matchCase ? "" : "i");
16 Object.defineProperty(this, "regexp", {value: regexp});
17 return regexp;
18
```

Figure 1 Translation code from AdBlock plus's rule to a regular expression.

Furthermore, ^ | at the end of a line is replaced by ^, which inherently includes a string terminator (denoted by $ in the corresponding regular expression).

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.eom/b.gif and http://ads.com/idx.html; however, ads.com^*.gif will filter only the former.

Throughout this paper, we only argue about AdBlock Plus because other extensions, such as uBlock, also take advantage of AdBlock Plus's filter notation.

3 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.

3.1 Machine Instructions

FARIS is abytecode interpreter. Thus, to perform pattern matching, AdBlock Plus's rules are translated into its machine instructions. FARIS interprets the four instructions 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 2 shows the instructions, operands, and what each instruction does (i.e., its semantics).

Table 2 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

Table 3 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 3 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 [44] 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 4.1.

3.2 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 4 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);

Table 4 Bytecodes of FARIS

Opcode	Bytecode	Priority
skip_scheme	0x83	high
match	0x84	high
char	if (! 0x80 & code)	low
skip_to	if (0x80 & code)	low

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.

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 3 shows these special characters.

4 Optimization and Implementation

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

4.1 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 2 shows how to

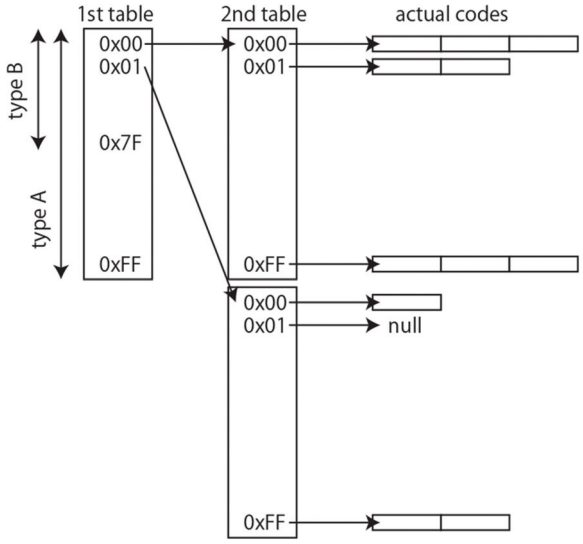


Figure 2 Prefix hash table.

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.

Note that the first table of type B needs only a range from 0x00 to 0x7F because * at the beginning of a line is previously eliminated, as described in Section 3.1. 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.

4.2 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.

4.3 GPGPU Optimization

Memory transfers between the CPU and GPU cause high overhead on GPGPU programs. Hence, we used only an array to represent all filter rules and transferred it from the CPU to GPU; this memory transfer is only required when filters are updated, which is infrequent. In addition, we adopted a mechanism of bulk query transfer to reduce the overhead of a GPGPU's kernel function call, which requires several thousand CPU clock cycles.

Furthermore, to mitigate the performance penalty caused by branch divergence [45–47], we sorted filter rules by a binary comparison of their bytecodes. By doing so, we could adjust the threads such that they tend to interpret the same instructions. Figure 3 intuitively illustrates how sorted and shuffled filter rules are executed. All the threads in a warp of GPU can simultaneously execute the same instructions. Thus, sorted filter rules are probably executed simultaneously on GPU, but shuffled filter rules are not.

4.4 Implementation

We implemented FARIS in C++ for CPU and GPGPU using NVIDIA's CUDA [48]. FARIS requires C++ compilers that support C++0x11; thus, CUDA version 7.0 or higher is required to compile for a GPGPU. We distributed our proof-of-concept implementation on the Web [15] as open source software under BSD licensing for scientific reproducibility; thus, anyone can use and modify it freely.

Figure 4 shows an example code for using FARIS. First of all, the header file of FARIS is included on line 1, and then the farisvm class is instantiated on line 7. Next, filter rules of AdBlock Plus are inserted to the instance on lines 10, and 11 by the add_rule function, which takes two arguments; a filter rule and a filter file. The result vector is defined for taking results of filtering

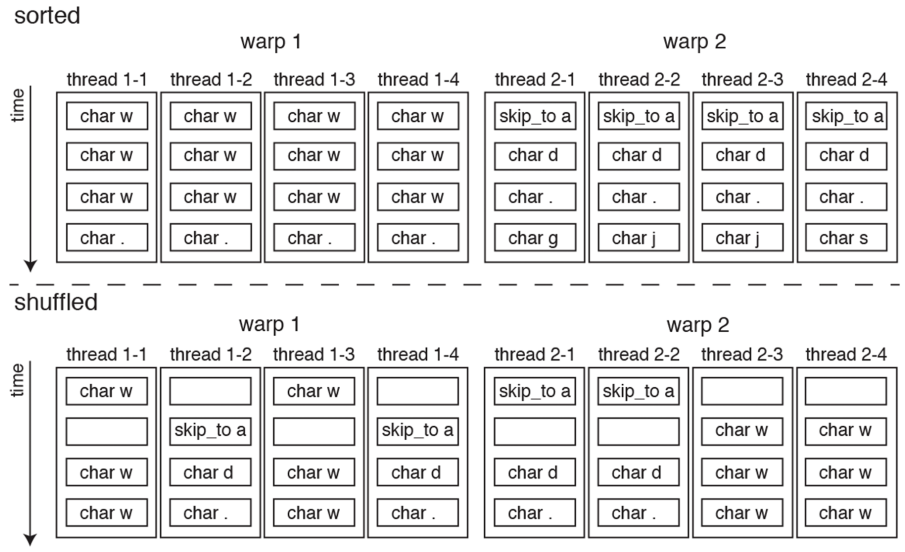


Figure 3 Sorted and shuffled filter rules on GPGPU.

```
1  #include "farisvm.hpp"
2  #include <iostream>
3
4  int
5  main(int argc, char *argv[])
6  {
7      farisvm vm;
8
9      // add filtering rules
10     vm.add_rule("||example.com^index", "filter1.txt");
11     vm.add_rule(".swf|", "filter2.txt");
12
13     // do matching
14     std::vector<farisvm::match_result> result[3];
15     farisvm::query_uri query[3];
16
17     query[0].set_uri("https://www.google.com/", "http://referer.com/");
18     query[1].set_uri("http://example.com/index.html", "http://referer.com/")
   ;
19     query[2].set_uri("http://example.com/index.swf", "http://referer.com/");
20
21     vm.match(result, query, 3);
22
23     for (int i = 0; i < 3; i++) {
24         std::cout << query[i].get_uri() << std::endl;
25         for (auto ret: result[i]) {
26             std::cout << "  rule: " << ret.rule
27                       << "\n  file: " << ret.file << std::endl;
28         }
29         std::cout << std::endl;
30     }
31
32     return 0;
33 }
```

Figure 4 Example code of FARIS.

on line 14, and the query array is defined for passing URIs to the filtering function on line 15. On lines 17, 18, and 19, query URIs are set by the set_uri function, which takes two arguments; a URI and an HTTP referrer.

Filtering is performed on line 21 by the match functions, which takes three arguments for results, queries, and the number of queries. Finally, from line 23 to 30, the results are printed to the standard output.

5 Evaluation

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

5.1 Dataset

For our evaluation, we captured real HTTP requests at the WIDE Camp 2015 Spring [56], 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.

Table 5 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 5 shows the top 30 most requested domains. There were several types of domains, including

Table 5 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

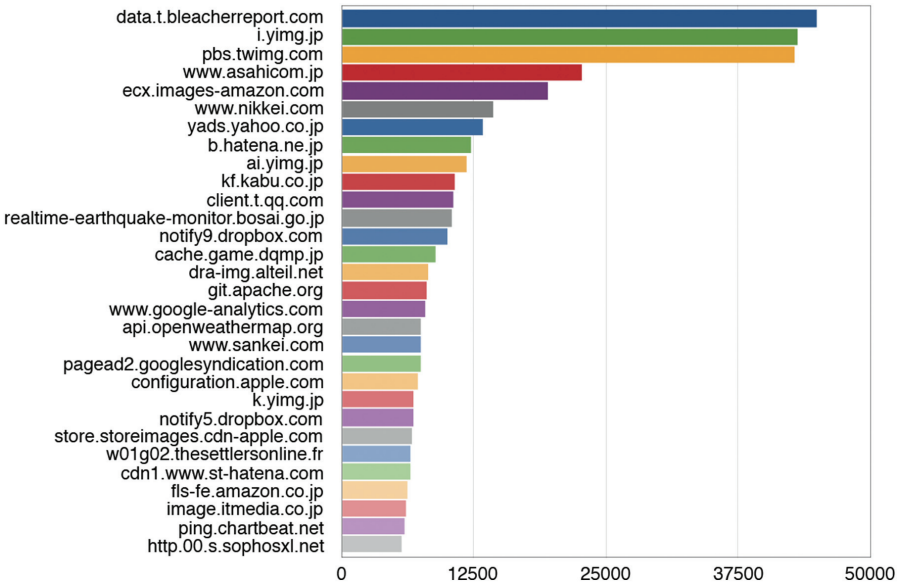


Figure 5 Histogram of top 30 requested domains.

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 6 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 6 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

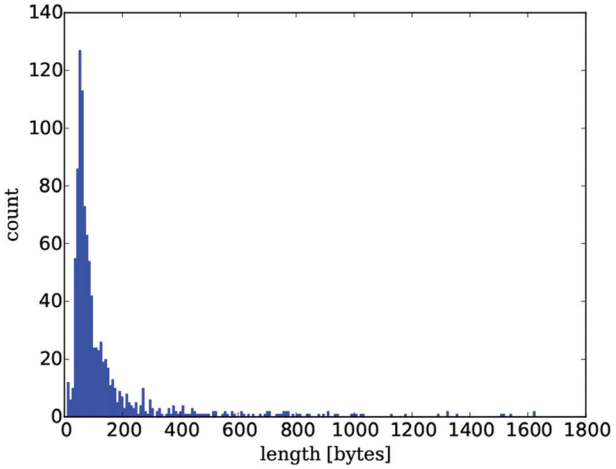


Figure 6 Histogram of URL length (1,000 samples).

Table 6 Filter statistics

	Filter Type		Instructions		
Filter	#url	#element	#char	#skip_to	#skip_scheme
easylist [11]	20,599	25,330	413,914	2,530	13,440
easyprivacy [49]	9,810	0	203,906	1,068	7,490
malware domains [50]	22,845	0	452,283	0	22,845
fanboy annoyance [51]	3,941	12,532	68,083	352	979
japanese [12]	9,600	0	127,302	481	5,063
japanese (tofu) [52]	1,511	352	24,471	84	1,102
easylist france [53]	2,856	1,645	72,424	652	2,548
easylist germany [54]	4,320	3,645	119,289	1,066	4,068
easylist italy [55]	1,657	1,221	47,436	438	1,596
total	77,139	44,725	1,529,108	6,671	59,131

on URL filtering. Table 6 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. Figure 7 shows a histogram of the length of filter rules denoted in FARIS's instructions. Here, 50% and 95% of the filters can be described as less than 20 and 41 instructions, respectively.

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 [11]; (2) a set of typical filters consisting of EasyList, EasyPrivacy [49], and malware domains [50]; and (3) a set of all filters consisting of every filter shown in Table 6.

5.2 Targets and Measurement Environment

Table 7 shows the evaluation targets of FARIS. There were four types of implementations: (1) an implementation without any optimization, as described in Section 4; (2) an implementation on a GPGPU; (3) an implementation

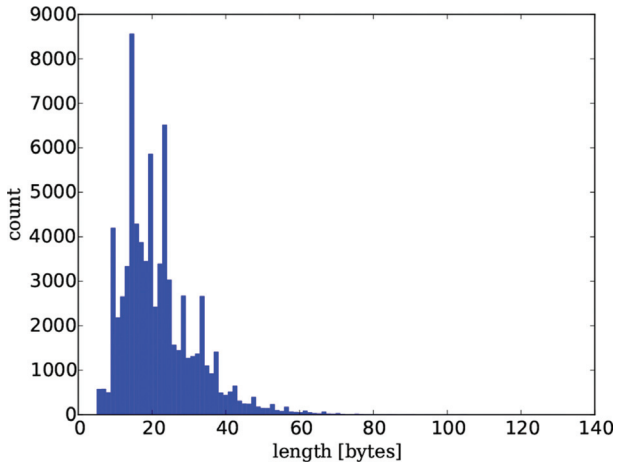


Figure 7 Histogram of filter lengths versus the number of FARIS instructions.

Table 7 Targets for evaluation

	Hash (Sec. 4.1)	MT (Sec. 4.2)	GPGPU
FARIS			
FARIS on GPGPU			o
hashed FARIS	o		
hashed FARIS (MT)	o	o	

with the prefix hash table described in Section 4.1; and (4) a multithreaded implementation with a prefix hash table.

Table 8 shows the measurement environments. As shown in the table, we conducted experiments on a MacBook Pro (for the CPU) and GPU server (for the GPGPU).

For comparison, we evaluated the performance of Irregexp [57] via V8 [58] and RE2 [59], which are regular expression engines. More specifically, we implemented filters of AdBlock Plus's filter rules in JavaScript and C++ using RE2.

5.3 Performance Evaluation

5.3.1 Memory usage

Figure 8 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.

5.4 Throughput and Execution Time

Figure 9 shows results of our throughput testing for FARIS and the regular expression engines. When using all filters, FARIS and FARIS on the GPGPU could manage 162 and 871 URLs per second, respectively, but RE2 and Irregexp managed only 17 and 7 URLs per second, respectively. These results

Table 8 Measurement environment

	MacBook Pro	GPU Server
CPU	Core i7 I7-4870HQ 2.5 [GHz], 4-core, HT turbo boost 3.7 [GHz]	Xeon E5-2609 v2 2.5 [GHz], 4-core x 2
GPU		Quadro K3100M
CUDA cores		768
OS	MacOS 10.10.2	Ubuntu 14.10 Linux Kernel 3.16.0-33
C++ compiler	llvm clang++ 6.0	g++ 4.9.1 (nvcc)
CUDA		7.0.28

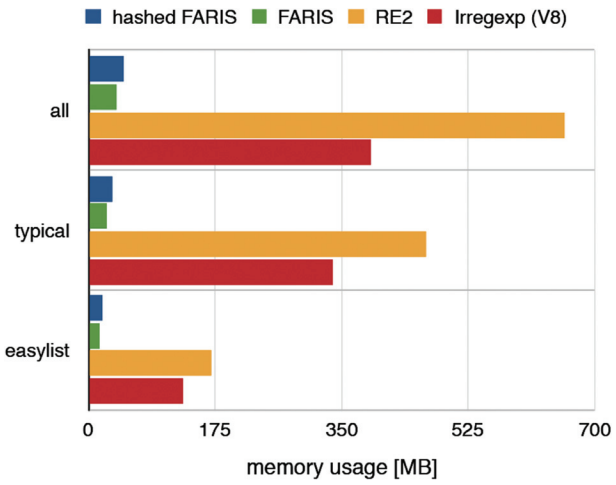


Figure 8 Memory usage.

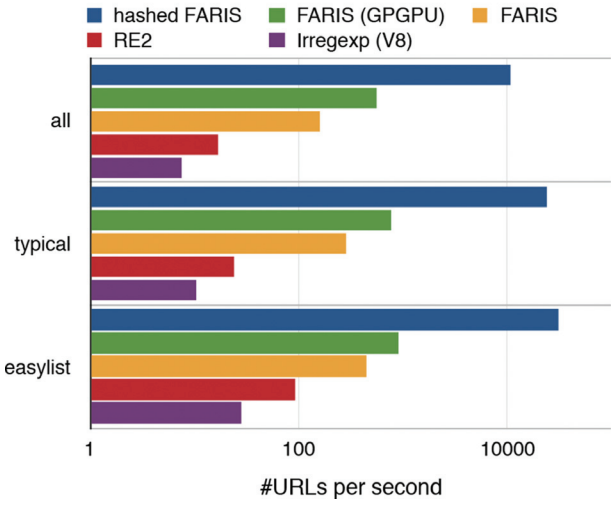


Figure 9 Matching throughput.

show that FARIS was 23 times faster than conventional implementations written in JavaScript. Furthermore, we observe that using the GPGPU was an efficient means to increase the throughput. Hashed FARIS far outperformed the others, managing 10,676 URLs per second, which was 1,423 times faster than the JavaScript implementation.

The throughput increases as the number of filter rules decreases in all cases except for FARIS on the GPGPU because of the lower GPU core utilization. To maximize GPU core utilization, an increased number of threads should be spawned on the GPU. However, our implementation launches threads corresponding to the number of filter rules; thus, only 20,599 threads were launched for EasyList, but $1{,}024 \times 768 = 786{,}432$ threads would be required to maximize the utilization of GPU cores when using the Quadro K3100M. To make matters worse, a small number of filter rules tended to incur branch divergence because of the decreasing number of similar rules.

Figure 10 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 12 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.

In Figure 10, we note the existence of many outliers on FARIS. The reason is based on the relation between the execution times and lengths of input URLs, as shown in Figure 11. As a consequence of the different execution times depending on the lengths of the input URLs, such outliers were produced. The outliers shown in Figure 12 exist for the same reason, as well as the non-uniform distribution of the prefixes of codes.

5.4.1 Optimization and performance on GPGPU

Figure 14 shows the throughput for when we adopted bulk query transfers for the GPGPU. For all filter sets, 20 simultaneous queries were the upper limit for effectively gaining throughput, with over 20 simultaneous queries barely affecting the throughput.

Figure 15 shows the throughput when shuffling and sorting filter rules for the GPGPU. FARIS on the GPGPU with sorted rules outperformed FARIS on the GPGPU with randomly shuffled rules, being approximately 2.6 times faster, thus revealing that sorting filter rules helps to mitigate the penalties incurred by branch divergence.

5.4.2 Multicore scalability

Figure 16 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

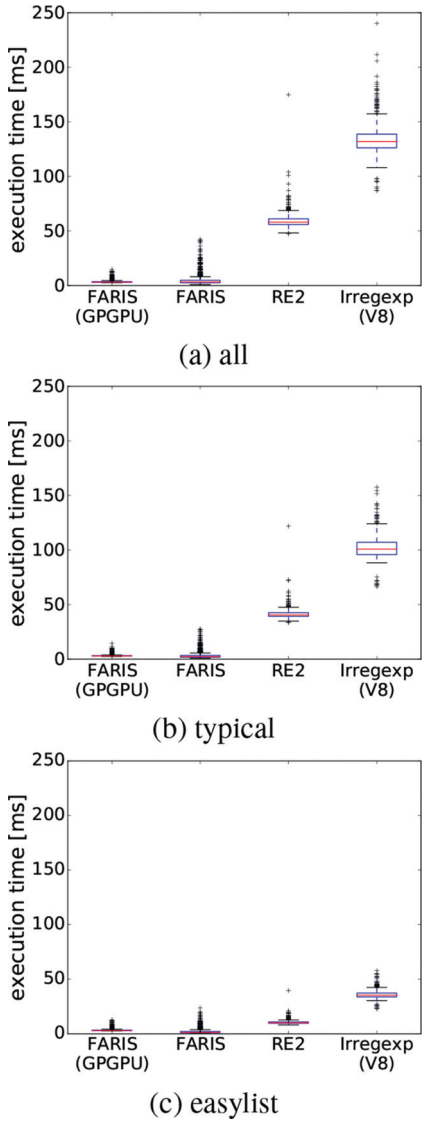
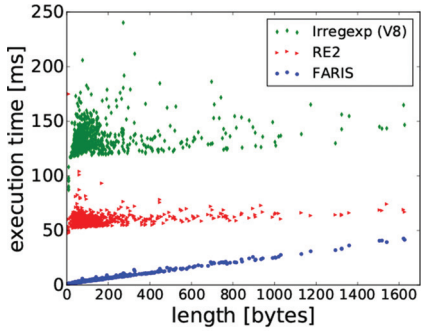
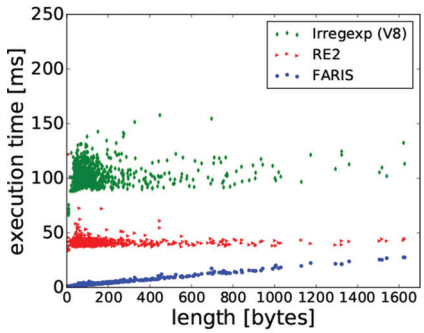


Figure 10 Execution times for matching URLs.

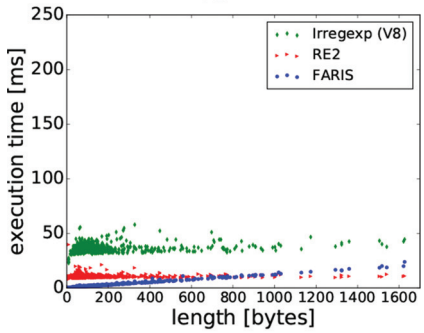
was equal to the number of threads. Note that the slope changed at four threads because of Intel's hyper-threading technology [60]. We confirmed that using more than eight threads did not contribute to the throughput but rather suffered from the overhead of thread scheduling and context switches.



(a) all



(b) typical



(c) easylist

Figure 11 Execution times versus URL lengths.

5.5 Filter and User Data Analysis

In this section, we show how AdBlock Plus's filter works on actual network traffic. Figure 17 shows filter rules and the number of filtered URLs captured at the WIDE camp. In total, we captured 1,760,898 URLs, of which 238,894

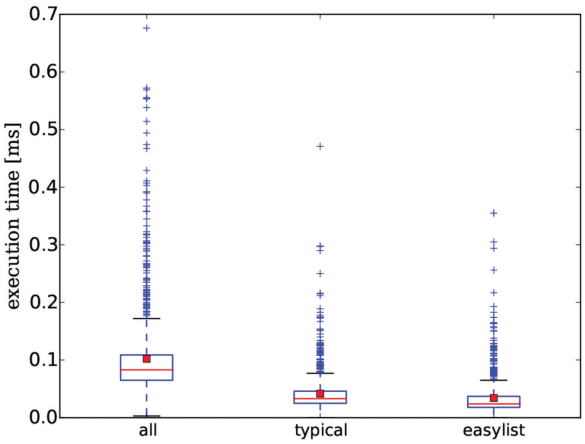


Figure 12 Execution times of Hashed FARIS for matching URLs.

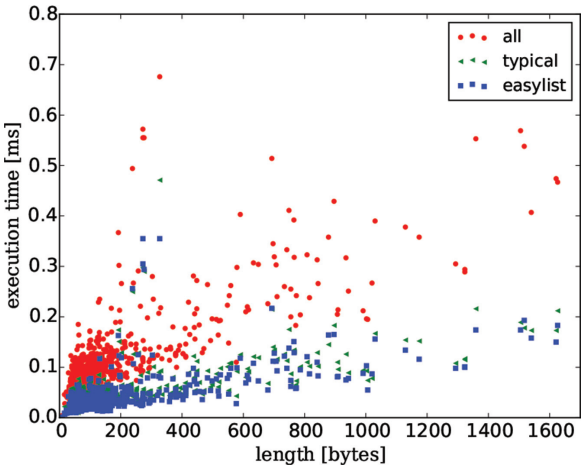


Figure 13 Execution times of Hashed FARIS versus URL lengths.

(13.7%) URLs were filtered by all filters. The japanese filter [12] was the most used filter; easylist [11], easylist privacy [49], and japanese filter (tofu) [52] were also used to filter many URLs.

Figure 18 shows a histogram of the filtered URLs, which included many advertisement domains not presented in Figure 5, such as doubleclick.net and ad.adlantis.jp. These results reveal that the filter should have properly worked to filter ad sites.

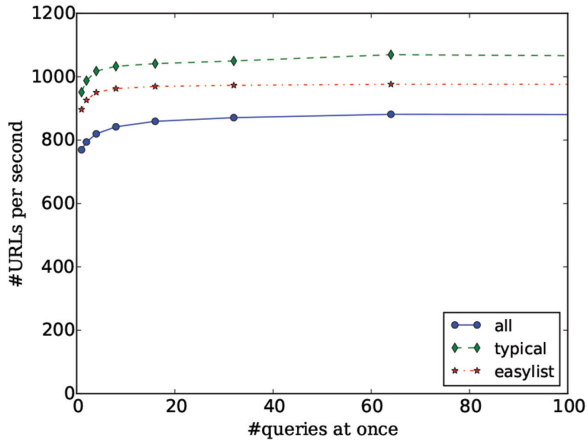


Figure 14 Matching throughput versus the number of simultaneous queries (GPGPU).

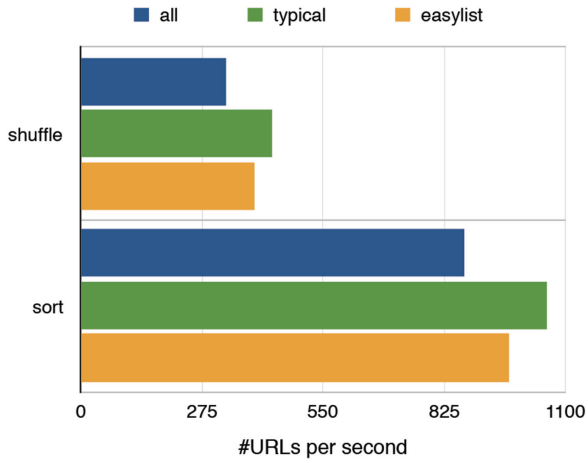


Figure 15 Matching throughput when shuffling and sorting filters (GPGPU).

Table 9 shows the rules actually used in our experiment. Despite many filter rules, only a few rules were actually used. For example, the filters of EasyList, EasyPrivacy, and malware domains had 20,599, 9,810, and 22,845 rules, respectively; however, only 78, 210, and 3 rules were actually used, respectively. This reveals that the number of filter rules can probably be reduced by adopting strategies of statistical analysis or collaborative filtering. However, reducing rules may lead to serious problems regarding malicious

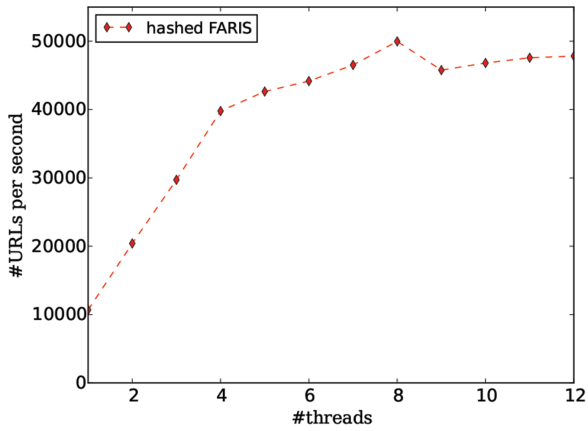


Figure 16 Matching throughput versus the number of threads (all filters).

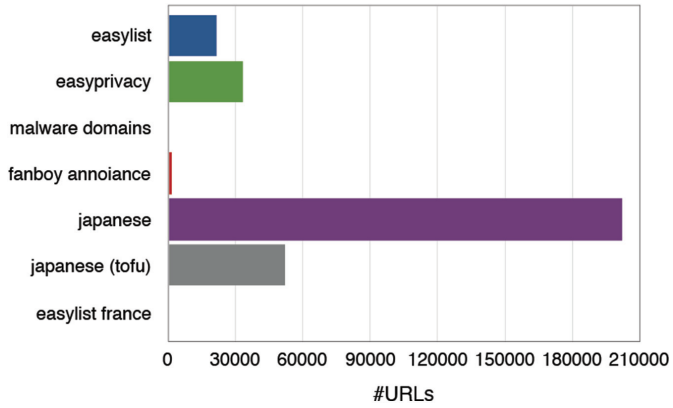


Figure 17 Filters and the number of filtered URLs.

traffic. For example, disabling the filtering of malware domains may allow unsuspected network traffic to intrude.

5.6 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 $O(\frac{L \times M}{T} + \alpha)$ on average, where T and α are constant

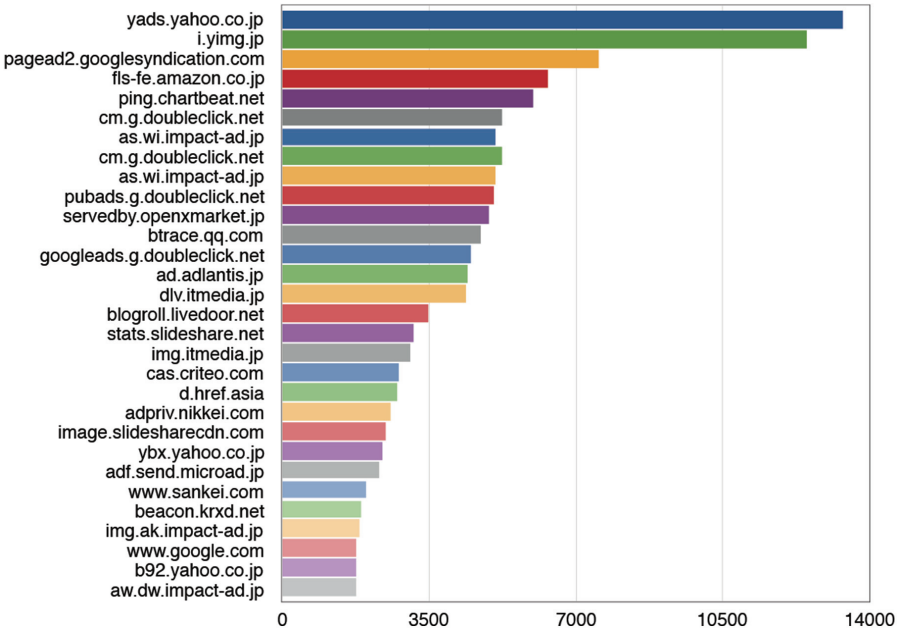


Figure 18 Histogram of the top 30 filtered domains.

Table 9 Rules used at WIDE camp 2015 spring

	#rules	Actually Used
easylist	20,599	78 (0.38%)
easyprivacy	9,810	210 (2.14%)
malware domains	22,845	3 (0.01%)
fanboy annoyance	3,941	44 (1.12%)
japanese	9,600	703 (7.32%)
japanese (tofu)	1,511	108 (7.15%)
easylist france	2,856	6 (0.21%)

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. FARIS on the GPGPU will help to speed up the exhaustive search under such unusual circumstances, but it should be unrealistic.

6 Discussion

In this section, we discuss filter rules and applications of FARIS.

6.1 Filter Rules

A key feature of FARIS is that it adopts AdBlock Plus's filters, which are widely used in Web browsers. To protect users from malicious Web or unwilling Web tracking sites, the freshness of the filters is an important factor. However, generally, the management cost for the filters tends to be expensive.

Thus, it is required a common filter rule to reduce the cost, and AdBlock's filter rule is currently one of the de facto standard for URL filtering. By using FARIS, the management cost could be reduced because it can take advantage of the filters of AdBlock Plus.

FARIS has the advantage mentioned above, but it has a disadvantage that regular expression cannot be handled. In AdBlock Plus or uBlock, AdBlock Plus's filters are translated into regular expressions, and they can thus deal with notations of regular expression. For example, "http://(www|ftp).ads" is a valid rule for AdBlock or uBlock, but it is invalid for FARIS. Fortunately, few filters on AdBlock Plus make use of the notations of regular expression.

Furthermore, compared to the hardware processing methods, FARIS has a drawback in terms of processing speed. Generally, hardware approaches, such as [24, 25], must be faster than the approaches for common CPUs, but hardware approaches consume much power and additional cost. In addition, these approaches cannot handle AdBlock's filter rule mentioned above.

6.2 Applications

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 dramatically increase AdBlock Plus's performance and reduce its large memory utilization. Thus, embedding FARIS into Web browsers orJavaScript engines is agood 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 [61]. Although user tracking and targeted advertisements are becoming the essence of free online services [10], they should definitely cause privacy concerns [6]. Phishing targeting on mobile devices [62] is also a significant threat. URL filtering should help to protect mobile device users from these threats.

Server applications and middleware such as firewalls or Web proxies can also adopt FARIS for filtering URLs; however, they may have sufficient memory or capabilities to implement other mechanisms [18, 22, 24]. Therefore, FARIS is not always the best solution for server applications or middleware which can make use of larage amount of memory or dedicated hardware such as FPGA.

7 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. We have distributed our proof-of-concept implementation on the website as open source software under BSD licensing for scientific reproducibility, and thus, anyone can use and modify it freely.

A key feature of FARIS is that it adopts AdBlock Plus's filters, which are widely used in Web browsers. Generally, the freshness of URL filters is important, but keeping fresh requires significant management cost. Therefore the ability that FARIS can take advantage of the filters of AdBlock Plus should be useful for management and deployment of URL filters.

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. Third, the GPGPU enables massively parallel computing, and therefore, we applied it to FARIS for sequential search.

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. We also revealed that the throughput of multithreaded FARIS is scalable on multiple CPU cores and that FARIS on the GPGPU was 5.2 times faster than FARIS on a single CPU core.

In summary, FARIS should be suitable for browser applications rather than server applications or middleware because FARIS is memory efficient and can take advantage of AdBlock Plus's filter rules, which are freely available on the Web. For the same reasons, it should be suitable for mobile devices, which have limited computational resources.

Acknowledgement

We would like to thank the WIDE project for supporting our experiments.

Appendix

8 AdBlcok Plus in C++11

Through our work, we observed that the regular expression library provided by C++11 was quite slow when implementing AdBlock Plus by using C++'s regex. Figure 19 shows the execution time of AdBclok Plus with the regular

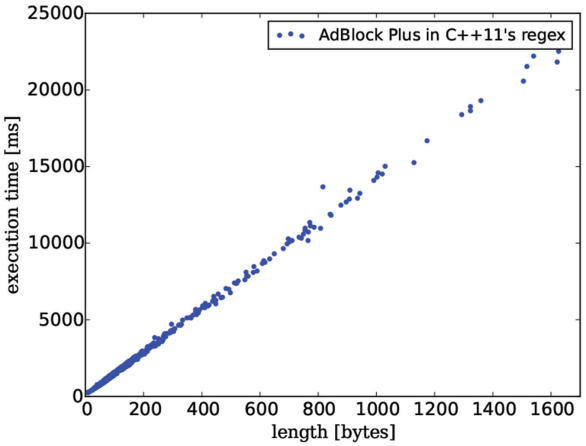


Figure 19 Execution time versus URL length (C++11).

expression library provided by C++11 (i.e., AdBlock Plus with C++11 regex). When using RE2, URL matching can be performed within 200 ms; however, AdBlock Plus with C++11 regex required 20,000 ms for matching in unusually bad cases. These results were significantly inefficient in comparison with other implementations. Therefore, we confirmed that the performance of a regular expression library depends on how it is implemented.

In this appendix, we discuss the reason why it was slow as well as a solution to this problem. Here, we evaluate ECMAScript's regular expression library provided by LLVM clang++ [63] April. 10, 2015)[1].

8.1 Methodology

In this section, we show the analysis of the library. We adopted static and dynamic analysis techniques to reveal the library's internal mechanism.

8.1.1 Understanding C++'s class and its hierarchy by static analysis

First, we analyzed the hierarchy of the classes by reading the source code and using Doxygen [64].

8.1.2 Understanding function calls by dynamic analysis

Then, we performed dynamic analysis using a debugger to understand the performance of the given function calls.

8.1.3 Obtaining information regarding stack frames and runtime-type information by dynamic analysis

Finally, we performed dynamic analysis and obtained information regarding stack frames and runtime-type information by embedding codes to obtain them. To accomplish this, we developed a library [65] for tracing stack frames and runtime-type information.

8.2 Results

Figure 20 shows the class hierarchy generated by Doxygen. We found that the __node class is used as the superclass to represent nodes of a finite state machine and derived classes from __node are used as actual nodes. Moreover, Figure 21 shows the matching states defined in __state structure. This analysis

[1] http://llvm.org/svn/llvm-project/libcxx/trunk/src/regex.cpp and http://llvm.org/svn/llvm-project/libcxx/trunk/include/regex

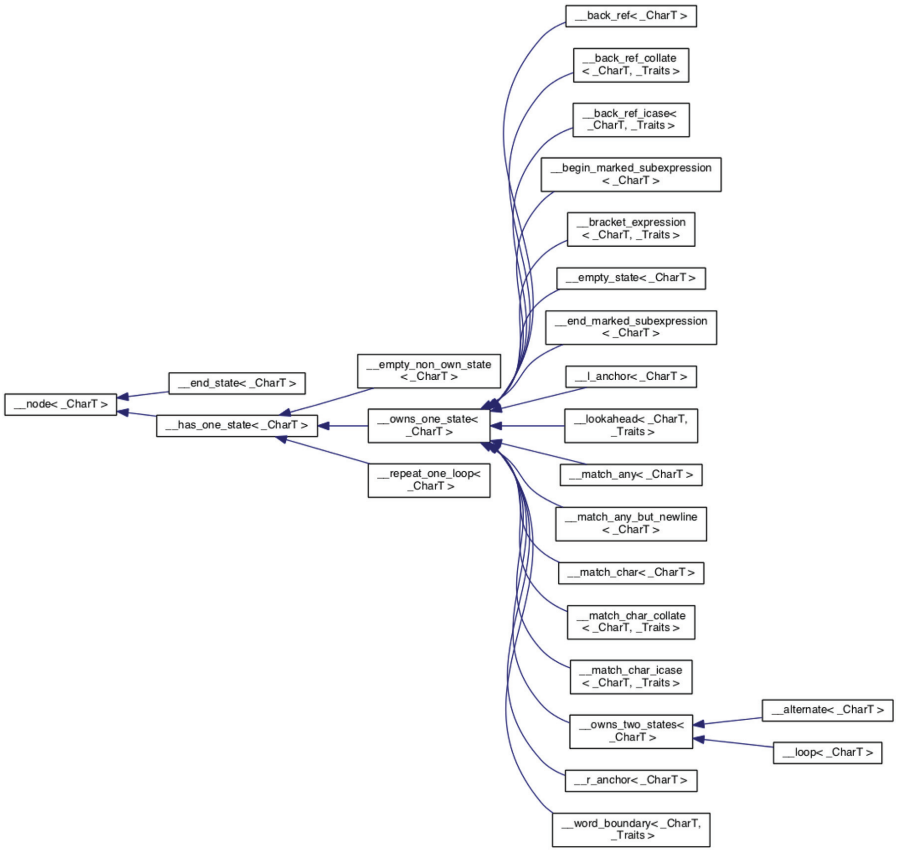


Figure 20 Class hierarchy of C++11's regex.

revealed that C++11 regex is based on a finite state machine. Furthermore, C++11 regex must be an NFA because there is a `__split` state in the code.

By conducting dynamic analysis, we found that the `__exec()` function defined in the __node class is a virtual function used to perform matching. Thus, we embedded codes into `__exec()` to obtain stack frames and runtime-type information. By analysing function calls, We found that C++11 regex adopts a depth-first search (DFS) algorithm for traversing an NFA.

8.3 Discussion

In [66], Rus Cox indicated that the use of a DFS algorithm for an NFA is inefficient. In this appendix, we revealed that C++11 regex adopts a DFS

```
template <class _CharT>
struct __state
{
    enum
    {
        __end_state = -1000,
        __consume_input,  // -999
        __begin_marked_expr, // -998
        __end_marked_expr,   // -997
        __pop_state,           // -996
        __accept_and_consume,  // -995
        __accept_but_not_consume,  // -994
        __reject,                  // -993
        __split,
        __repeat
    };
    /* omitted */
};
```

Figure 21 States of C++11's regex.

algorithm and confirmed that it is rather slow. In [67], Ken Thompson showed that a width-first search (WFS) algorithm on an NFA works relatively much more efficiently. Therefore, we conclude that a WFS algorithm should be adopted for traversing the given NFA; however, C++11 regex does not use such an approach.

To replace the DFS algorithm of the C++11 regex library with the WFS algorithm, matching must be concurrently performed at the `__exec_split()` function or when in state `__state::__split`. This could be achieved by preparing a buffer of pointers to nodes and locations within the given input string.

References

[1] Privoxy (2016). Available at: http://www.privoxy.org/ [accessed June 26, 2016]
[2] SquidGuard (2016). Available at: http://www.squidguard.org/ [accessed June 26, 2016]
[3] Squid (2016). Available at: http://www.squid-cache.org/ [accessed June 26, 2016]
[4] AdBlock Plus (2016). Available at: https://adblockplus.org/ [accessed June 26, 2016]
[5] uBlock (2016). Available at: https://www.ublock.org/ [accessed June 26, 2016]

[6] Mayer, J. R., and Mitchell, J. C. (2012). "Third-Party web tracking: policy and technology," *Proceedings of the IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012*, (San Francisco, CA: IEEE Computer Society), 413–427.

[7] Roesner, F., et al., (2012). "Detecting and defending against third-party tracking on the web," *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012*, eds S. D. Gribble and D. Katabi, (San Jose, CA: USENIX Association), 155–168.

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

[9] EasyList (2016). Available at: https://easylist.adblockplus.org/en/ [accessed June 26, 2016]

[10] Gill, P., et al., (2013). "Best paper – Follow the money: understanding economics of online aggregation and advertising," in *Proceedings of the 2013 Conference on Internet measurement Conference* eds Papagiannaki et al. (New York, NY: ACM), 141–148.

[11] EasyList (AdBlock Plus Filter) (2016). Available at: https://easylist-downloads.adblockplus.org/easylist.txt [accessed June 26, 2016]

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

[13] Writing AdBlock Plus Filters (2016). Available at: https://adblockplus.org/en/filters [accessed June 26, 2016]

[14] Cox, R. (2007). *Regular Expression Matching: The Virtual Machine Approach.* Available at: https://swtch.com/∼rsc/regexp/regexp2.html

[15] Takano, Y. (2016). *FARIS: Fast and Memory-effcient URL Filter.* Available at: https://github.com/starbed/farisvm [accessed June 26, 2016]

[16] Knuth, D. E., et al., (1977). Fast pattern matching in strings. *SIAM J. Comput.* 6, 323–350.

[17] Boyer, R. S., and Moore, J. S. (1977). A fast string searching algorithm. *Commun. ACM* 20, 762–772.

[18] Aho, A. V., and Corasick, M. J. (1975). Effcient string matching: an aid to bibliographic search. *Commun. ACM* 18, 333–340.

[19] Michel, B. S., et al. (2000). "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, 670–678.

[20] Garnica, J. J. et al. (2012). "A FPGA-based scalable architecture for URL legal filtering in 100GbE networks," in *Proceedings of the 2012 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2012*, Cancun, 1–6.

[21] Morrison, D. R. (1968). PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 514–534.

[22] Zhou, Z. et al. (2010). "A high-performance URL lookup engine for URL filtering systems," in *Proceedings of IEEE International Conference on Communications, ICC 2010*, Cape Town, 1–5.

[23] Wu, S., and Manber, U. (1994). *A Fast Algorithm for Multi-Pattern Searching*.

[24] Huang, N., et al. (2005). "A fast URL lookup engine for content-aware multi-gigabit switches," in *19th International Conference on Advanced Information Networking and Applications (AINA 2005)*, Taipei, 641–646.

[25] Pagiamtzis, K., and Sheikholeslami, A. (2006). "Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey," *IEEE J. Solid-State Circuits* 41, 712–727.

[26] Feng, Y., et al. (2011). "An effcient caching mechanism for network-based URL filtering by multi-level counting bloom filters," in *Proceedings of IEEE International Conference on Communications, ICC 2011*, Kyoto, 1–6.

[27] Fan, L., et al. (2000). Summary cache: a scalable wide-area web cache sharing protocol. *IEEE ACM Trans. Netw*. 8, 281–293.

[28] Bonomi, F., et al. (2006). "Beyond bloom filters: from approximate membership checks to approximate state machines," in *Proceedings of the ACM SIGCOMM 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Pisa, eds L. Rizzo, T. E. Anderson and N. McKeown, 315–326.

[29] Bonomi, F., et al. [2006]. "An improved construction for counting bloom filters," in *Proceedings of the 14th Annual European Symposium*, ed. Y. Azar and T. Erlebach, (Zurich: Lecture Notes in Computer Science), 11–13.

[30] Yuan, Z., et al. (2013). "TFD: A multi-pattern matching algorithm for large-scale URL filtering," in *Proceedings of the International Conference on Computing, Networking and Communications, ICNC 2013*, San Diego, CA, 28–31.

[31] Aoe, J. (1989). An Effcient digital search algorithm by using a double-array structure. *IEEE Trans. Softw. Eng*. 15, 1066–1077.

[32] Vasiliadis, G. et al. (2008). "Gnort: High Performance Network Intrusion Detection Using Graphics Processors," in *Proceedings of the Recent Advances in Intrusion Detection, 11th International Symposium*, eds R. Lippmann, E. Kirda, and A. (Trachtenberg Cambridge, MA: Lecture Notes in Computer Science), 116–134.

[33] Wang, L., et al. (2011). "Gregex: GPU based high speed regular expression matching engine," in *Proceedings of the Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS* eds I. You, L. Barolli, F. Tang, and F. Xhafa, (Seoul: IEEE Computer Society), 366–370.

[34] Cascarano, N., et al. (2010). iNFAnt: NFA pattern matching on GPGPU devices. Comput. Commun. Rev. 40, 20–26.

[35] Zu, Y. et al. (2012). "GPU-based NFA implementation for memory effcient high speed regular expression matching," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, eds J. Ramanujam and P. Sadayappan (New Orleans, LA: ACM), 25–29.

[36] Yu, X., and Becchi, M. (2013). "GPU acceleration of regular expression matching for large datasets: exploring the implementation space," in *Proceedings of the Computing Frontiers Conference, CF'13*, eds H. Franke, A. Heinecke, K. V. Palem, and E. Upfal, (Ischia: ACM), 18.

[37] Chou, L. D. et al. (2012). "Design and Implementation of Content-based Filter System on Embedded Linux Home Gateway," in *Proceedings of the 14th International Conference on Advanced Communication Technology (ICACT)*, PyeongChang.

[38] Wang, A., et al. (2013). "SOBA: a services-oriented browser architecture with distributed url-filtering mechanisms for teenagers," in *Proceedings of the IEEE Ninth World Congress on Services*, (Santa Clara, CA: IEEE Computer Society), 67–74.

[39] Dalek, J. et al. (2013). "A method for identifying and confirming the use of URL filtering products for censorship," in *Proceedings of the 2013 Conference on Internet Measurement Conference*, (New York City, NY: ACM), 23–30.

[40] Ma, J., et al., (2009). "Beyond blacklists: learning to detect malicious web sites from suspicious URLs," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, June 28 – July 1* (New York, NY: ACM), 1245–1254.

[41] Ma, J., et al. (2009). "Identifying suspicious URLs: an application of large-scale online learning," in *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14–18, 2009*, Vol. 382, ed. A. P. Danyluk et al. (New York, NY: ACM), 681–688.

[42] Ma, J. et al. (2011). Learning to detect malicious URLs. *ACM Trans. Intell. Syst. Technol.* 2, 30.

[43] Su, K., et al. (2013). "Suspicious URL filtering based on logistic regression with multi-view analysis," in *Proceedings of the Eighth Asia Joint Conference on Information Security, Asia JCIS 2013, Seoul, Korea, July 25–26, 2013* (Rome: IEEE), 77–84.

[44] Berners-Lee, T., Fielding, R., and Masinter, L. (2005). *"Uniform Resource Identifier (URI): Generic Syntax," RFC 3986 (INTERNET STANDARD), Jan. 2005. Updated by RFCs 6874, 7320*. Available at: https://www.rfc-editor.org/rfc/rfc3986.txt

[45] Fung, W. W. L., and Aamodt, T. M. (2011). "Thread block compaction for effcient SIMT control flow," in *Proceedings of the 17th International Conference on High-Performance Computer Architecture (HPCA-17 2011), February 12–16, 2011* (San Antonio, TX: IEEE Computer Society), 25–36.

[46] Narasiman, V., et al. (2011). "Improving GPU performance via large warps and two-level warp scheduling," in *Proceedings of the 44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, 3–7 December 2011, Porto Alegre, Brazil*, ed. C. Galuzzi, L. Carro, A. Moshovos and M. Prvulovic (New York, NY: ACM), 308–317.

[47] Han, T. D., and Abdelrahman, T. S. (2011). "Reducing branch divergence in GPU programs," in *Proceedings of 4th Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2011, Newport Beach, CA, USA, March 5, 2011* (New York, NY: ACM), 3.

[48] NVIDA CUDA ZONE. Available at: https://developer.nvidia.com/cuda-zone

[49] EasyPrivacy (2016). *EasyPrivacy (AdBlock Plus Filter)*. Available at: https://easylist-downloads.adblockplus.org/easyprivacy.txt

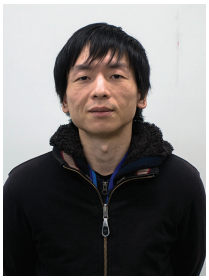
[50] Malware Domains (2016). *Malware Domains (AdBlock Plus Filter)*. Available at: https://easylist-downloads.adblockplus.org/malwaredomains_full.txt

[51] Funboy Annoyance (2016). *Funboy Annoyance (AdBlock Plus Filter)*. Available at: https://easylist-downloads.adblockplus.org/fanboy-annoyance.txt

[52] Japanese Tofu Filter (2016). *Japanese Tofu Filter (AdBlock Plus Filter)*. Available at: http://tofukko.r.ribbon.to/Adblock_Plus_list.txt
[53] EasyList France (2016). *EasyList France (AdBlock Plus Filter)*. Available at: https://easylist-downloads.adblockplus.org/liste_fr.txt
[54] EasyList Germany (2016). *EasyList Germany (AdBlock Plus Filter)*. Available at: https://easylist-downloads.adblockplus.org/easylistgermany.txt
[55] EasyList Italy (AdBlock Plus Filter) (2016). Available at: https://easylist-downloads.adblockplus.org/easylistitaly.txt [accessed June 26, 2016]
[56] WIDE Project (2016). Available at: http://www.wide.ad.jp/ [accessed June 26, 2016]
[57] Irregexp, Google Chrome's New Regexp Implementation (2009). Available at: http://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html [accessed February 4, 2009]
[58] V8 JavaScript Engine (2016) Available at: https://chromium.googlesource.com/v8/v8.git [accessed June 26, 2016]
[59] Cox. R. (2016). *RE2*. Available at: https://github.com/google/re2 [accessed June 26, 2016]
[60] Intel®Hyper-Threading Technology (2016). Available at: http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html [accessed June 26, 2016]
[61] Felt, A. P. et al. (2011). "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*, eds X. Jiang, A. Bhattacharya, P. Dasgupta, and W. Enck (Chicago, IL: ACM), 3–14.
[62] Felt, A. P., and Wagner, D. (2011). "Phishing on mobile devices," in *Proceedings of the W2SP, 2011*.
[63] *clang: a C language family frontend for LLVM*. http://clang.llvm.org/
[64] *Doxygen: Main Page*. Available at: http://www.stack.nl/∼dimitri/doxygen/.
[65] *dghelper-cpp11*. Available at: https://github.com/myu2/dghelper-cpp11.
[66] Cox, R. (2007). *Regular Expression Matching Can Be Simple And Fast (but is Slow in Java, Perl, PHP, Python, Ruby, ...)* Available at: https://swtch. com/∼rsc/regexp/regexp1.html
[67] Thompson, K. (1968). Programming techniques: Regular expression search algorithm. *Commun. ACM* 11, 419–422, June 1968.
[68] Papagiannaki, K., Gummadi, P. K., and Partridge, C. (eds) (2013). in *Proceedings of the 2013 Internet Measurement Conference, IMC 2013*, (Barcelona: ACM).

Biographies

Y. Takano received B.E. from National Institution for Academic Degrees and Quality Enhancement of Higher Education, Japan, in 2003. He also received M.S. and Ph.D. in Computer Science from Japan Advanced Institute of Science and Technology, Japan, in 2005 and 2011, respectively. He is currently a researcher of National Institute of Information and Communications Technology, Japan. His research interests include computer systems, computer networks, and network security.



R. Miura received B.E. degree in Faculty of Industrial Science and Technology from Tokyo University of Science, Japan, in 2002. He also received M.S. in Computer Science from Japan Advanced Institute of Science and Technology, Japan, in 2007. He is currently a technical researcher of National Institute of Information and Communications Technology, Japan. His research interests include computer systems and network security.