

Automatic Ad Blocking: Improving AdBlock for the Mozilla Platform

Justin Crites and Mathias Ricken

Dept. of Computer Science

Rice University

Houston, TX 77005

xiphoris@rice.edu, mgricken@rice.edu

ABSTRACT

Advertising on the world wide web is increasingly becoming problematic. In addition to merely being a nuisance, advertisements may also pose a threat to user security. Blocking web advertisements can thus substantially improve the user's browsing experience. AdBlock is an open-source plugin for Mozilla browsers that uses regular expressions, an advanced concept unfamiliar to novice users, to block a limited number of HTML elements. This paper describes some of AdBlock's shortcomings and the remedies we implemented: automatically generating regular expressions, allowing for web updates, and vastly extending the set of blockable elements.

1. INTRODUCTION

In the last few years, the number of users accessing the world wide web has increased by a factor of 50. In 1995, only an estimated 16 million users surfed the web; this number has grown to over 800 million today. With about 12.7 percent of the world population and more than half of North America and Australia online, it is clear that the web has become an important marketplace [1].

In addition to the vast number of potential customers, delivery advantages provide an additional incentive for companies to advertise on the web: While the response rate for online advertisement is lower than the rate for traditional paper-based ads, for example, this disadvantage is offset by significantly lower costs. Online advertisements can also be delivered to customers all over the world at the same cost, tailored to particular customer groups, and be more interactive and entertaining. For all of these reasons, businesses have massively increased their online advertising expenditures, and will probably continue to do so. According to the Interactive Advertising Bureau, companies have spent USD 2.43 billion on web-based advertisement in the third quarter of 2004 alone, an

increase of 34 percent over the same quarter in 2003 (USD 1.70 billion). In 2004, online ads will thus account for 8 percent of the total advertising budget, and the American Advertising Federation predicts this percentage to double by 2007, thus supplanting other forms of advertising like magazine and radio ads [2].

Unfortunately, users will not only have to cope with an increased number of ads on the web, these ads will also grow in size and become more invasive and therefore potentially dangerous. Internet advertiser DoubleClick expects there will be fewer of the small banners that are currently in use, and more ads that fill up most of the screen. Furthermore, a larger proportion of online ads will use "rich" media like Flash, Shockwave, or popups. DoubleClick forecasts that these rich ads will surpass image ads for the first time in 2005 [3].

Online ads do not only fill up a user's screen, they may also undermine user security and privacy. Richer ads can often make use of more computer features, thus putting the user at a higher risk than a plain image advertisement could. But even a mere image can violate a user's privacy by exposing the computer's IP address to another party, which may allow that faction to track the user's surfing habits.

It is not surprising that several attempts are being made to block the appearance of online advertisement on user screens. One of these project is AdBlock [4], an open-source plugin for Mozilla browsers [5]. Since it is available free of charge, it has quickly gained popularity; however, the product is neither very powerful nor easy to use.

Section 2 will discuss the AdBlock plugin, its capabilities and deficiencies. Section 3 describes some important issues of developing extensions for the Mozilla platform, issues that often hindered us in quickly and elegantly implementing our

improvements to Adblock, which section 4 will explain in detail. The paper will conclude with a review in section 5.

2. THE ADBLOCK PROJECT

Adblock is an extension for Mozilla browsers that allows users to filter web content by blocking a small number of HTML tags that are often used for advertising purposes, such as images, embedded objects, and internal frames. These elements are often used legitimately, too, so indiscriminate blocking of all of their occurrences is not acceptable. To distinguish ads from normal content, Adblock uses a “blacklist”, which enumerates the source addresses, or URLs, of elements a user does not want to see.

Since listing all advertisements individually is ineffective, Adblock makes use of the fact that much of the offensive material originates from similar URLs and allows similar URLs to be blocked with just one “wildcard” entry in the blacklist. Such an entry is a simplified regular expression, specifying only parts of the URL in detail and denoting places where differences may occur with an asterisk (“*”). For example, if a user wanted to block JPEG images coming from both “somedomain.com” and “anotherdomain.com”, this could be expressed using the wildcard string “*domain/*.jpg”. Both the substrings “some” and “another” are admissible in place of the first wildcard, and anything may be inserted for the second wildcard, as long as the entire address still ends with “.jpg”.

Unfortunately, the use of regular expressions to specify ad URLs is an advanced concept that many novices do not understand. Since users with a weaker understanding of computing concepts probably fall prey to malicious advertisement more often, empowering novices to effectively block advertisement would be a very desirable accomplishment. Furthermore, entering regular expressions by hand is time-consuming and error-prone even for seasoned users.

Even with wildcarded strings, the creation of the blacklist remains one of the inefficient aspects of Adblock. Although many advertisements come from the same sources, it is likely that every user will go through the process of entering the very same URLs into the blacklist. Adblock already offers an import/export facility for blacklist entries; however, it requires the use of the computer’s file system, which

often proves problematic for beginners. As a tool which owes its entire existence to the world wide web, Adblock does not realize the full potential of the network: Instead of merely blocking web content, Adblock ought to use the web as delivery vehicle for updated blacklist entries.

Finally, Adblock is capable of blocking only a small set of HTML elements, namely , <EMBED>, <OBJECT>, and <IFRAME>. While the majority of online advertisement uses these tags, other forms of advertising are spreading, e.g. text ads. Since text ads do not use any of these four tags, Adblock is unable to remove them from a web page. The ability to remove any kind of tag – and all the elements contained in it – would create the option of removing text ads or other annoying features as well. Additionally, websites often place a number of ads in a table. Removing the table in its entirety is more powerful than painstakingly removing every single advertisement contained in it. With its current design, though, Adblock cannot make effective use of such ad groupings.

These three problems – regular expressions, inadequate sharing of blacklists, and restrictive set of blockable tags – were reason enough to examine ad blocking in more detail and to provide the Mozilla community with an improved product.

3. DEVELOPING FOR MOZILLA

Mozilla is an open-source platform for loading, displaying, and modifying XML pages and is written in C and C++. All user interaction, however, everything on the shiny surface of an application, is relegated to so-called “chrome providers”, extensions that are written in a combination of JavaScript and XUL, a proprietary XML user interface description language. By describing the entire user interface this way, including menus, toolbars, status bars, and dialogs, applications run on all systems to which the Mozilla platform has been ported.

Chrome providers can modify different portions of Mozilla and the content it processes, e.g. by changing the look-and-feel, adapting Mozilla to another system, translating Mozilla to another spoken language, or affecting the way content is processed and displayed. Quite naturally, Adblock does the latter and thus is a content provider for Mozilla.

The Mozilla Organization is quick to point out some of the benefits of their architecture. In addition to the increased portability mentioned above, the design also establishes a separation between trusted code, i.e. Mozilla itself, and extension code running in a sandbox. When examining the Mozilla platform and Adblock, however, we have experienced a number of disadvantages as well that substantially affect the way software is developed for Mozilla.

3.1. JAVASCRIPT PROBLEMS

A large portion of Mozilla applications is written in JavaScript, a language that lacks several features other programming languages provide. While the Mozilla developers have done a formidable job integrating JavaScript into their application, they do not provide an integrated development environment or any decent debugging support for extensions. Mozilla Firefox does have a “JavaScript Console” that displays some error messages, but it is clearly targeted at JavaScript run from regular web pages as opposed to code executed in an extension. Only in very rare circumstances did the console display any error message during our development, and even then was its output limited to unhelpful reports such as “Function not defined” without giving any hint why that was the case. It apparently is not possible to set a breakpoint in extension code and step through the code using an interactive. The only interactive tool Mozilla provides is the DOM Inspector, which allows the developer to examine the values of document object model (DOM) nodes at runtime.

The most common result of a programming error, even a simple syntax error like a missing brace, was quietly stopping the JavaScript interpreter, rendering most of Mozilla Firefox unresponsive. With such all-or-nothing behavior and virtually no information, debugging degenerated into trial and error. After scanning the code for obvious mistakes, we mostly resorted to divide and conquer, partially undoing the changes we had made to limit the number of lines that could potentially contain the problem. Another approach was to litter the code with calls to the `alert` function to display a value in a message box, the GUI equivalent of `printf` debugging. This, however, was not even possible all the time, since dialogs are rendered using XUL and JavaScript and sometimes caused infinite recursion. Writing to a log file or the user preferences mostly solved this particular issue.

In a few situations, an error did not terminate JavaScript interpretation but caused the interpreter to leave the method at the place of the error. These errors were particularly hard to find since a method call appeared to complete successfully, but failed to execute the entire method body, leading to inconsistent data.

Aside from the problems of properly supporting development using JavaScript, the very choice of that language seems somewhat questionable. Since JavaScript is prevalent among web scripting languages, it obviously has to be supported by Mozilla, but it was intended as a lightweight means of making the web interactive, not as a tool for developing entire applications. As such, it lacks strict typing and exhibits some other idiosyncrasies unknown in many other languages.

Variables in JavaScript do not have a static type assigned to them. There is no way to ensure at compile time that arguments to a function, for example, have the expected types. Mistakenly using a method an object does not support generates a runtime exception, which at the extension level often does not even get reported. This problem can be partially mitigated by programmatically asserting the correct types at the beginning of a method and displaying an error message if incorrect types are passed, but it nonetheless tedious and unsatisfactory.

The way variables are introduced and scoped also differs from many industrial-strength languages. Variable declarations are not necessary, which opens up the possibility for uncaught misspellings. During the development of our Adblock improvements, for example, we used a variable `pipeIndex`, but accidentally spelled it `pineIndex` once. Since this variable had not been used anywhere else, its value defaulted to `null`. This error, though trivial, was not found until much later and caused an unnecessary delay of the development process.

New scopes for variables are apparently only introduced when the `var` keyword is used to actually declare an untyped variable. As stated before, this is not necessary and not enforced in any way. If the `var` keyword is forgotten, JavaScript seems to access the a variable with the same name that was last accessed before, which might even appear to be a “local” variable in another function. The concepts of local and global variables, shadowing, and information

hiding hence get blurred. In an error similar to the one described above, we accidentally used the wrong variable and changed its value in the calling method, causing an unexpected side effect. It is certainly possible to get used to this system of variable usage, but the language semantics have to be explained in detail and debugging support needs to be improved.

3.2. PLATFORM PROBLEMS

Our biggest gripe with the Mozilla platform itself is the lack of documentation. Creating the platform documentation is certainly an ongoing process and its quality is due to improve over time, but incomplete specifications and descriptions are currently a huge impediment to efficient development. Online references for many JavaScript objects do exist but often provide nothing more than a list of method and property names devoid of any explanation. This leaves much to experimentation, which often is not intuitive and fruitful due to similar identifiers such as `name`, `nodeName`, `className`, `tagName` and `localName` existing all in one class. It is also very evident that the documentation is again geared towards website use as opposed to extension development, just like JavaScript. Many parts of the Mozilla DOM used to represent XML is well documented, but once extension developers need to deal with events, observers, browsers, or how to access the DOMs of other documents, the current documentation will fail them. The use of the DOM Inspector has proved vital in these areas and at this time.

The precise features of XUL, Mozilla's user interface description language, are also not well documented. For GUI development on a small scale, this is not a large issue, and the XUL tags are generally intuitive and easy to use, but a complete description of all the flags is a necessity for designing good interfaces. In general, though, XUL is an interesting approach and should be promoted and standardized.

Since XUL and JavaScript are used for extensive parts of the development, different parts of the same program are often distributed among several files. If the developers are not careful, this often obscures the control flow. The use of many different files, combined with JavaScript's scoping and sandboxing, also complicates reusing code and communicating between different parts of the program.

Finally, we express the hope that Mozilla will soon complete the implementation of their document object model and support all events described in the W3C specifications [6]. The Mozilla platform already implements those specifications more closely than any other platform of which we are aware, but several DOM mutation events are still missing, for example, and their lack has prevented us thus far from implementing our Adblock improvements as well as we desire. The current state of Mozilla's DOM and event system is commendable, but for a system so dependent on extensions as Mozilla, it is crucial to provide extensions with all the events and system hooks.

4. IMPROVEMENTS TO ADBLOCK

We decided to make Adblock both easier to use and more powerful by focusing on the three deficiencies already mentioned in section 2. Below, we describe (i) an algorithm for automatically generating wildcarded URLs and its implementation, (ii) an extension that allows for web updates of the blacklist, and (iii) an algorithm for blocking arbitrary HTML page sections and its implementation.

4.1. AUTOMATIC GENERATION OF WILDCARDED URLS

After determining that creating wildcarded strings and dealing with regular expressions are beyond the understanding of beginning computer users, we determined to reduce ad blocking to just a few mouse clicks while maintaining Adblock's efficacy. To describe our approach from the user's perspective, we coined the term "two-click blocking": We wanted to enable users to invoke effective ad blocking with just two mouse clicks – one to bring up a context menu, one to select automatic ad blocking. The program should then automatically generate wildcarded URLs as appropriate.

Algorithm and Implementation

When searching for ways to implement this, we considered Bayesian filtering; however, we discard that option once we found an existing project, AdblockLearner [7], that already attempted this with limited success. AdblockLearner is also a stand-alone program and thus not easy to use for novices, the user group at which this improvement is directed.

We decided to approach the task from more classical pattern matching perspective: Given two strings, we

j	0	1	2	3	4	5	6	
i			B	D	C	A	B	A
0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

Figure 1: Dynamic programming algorithm for LCS.

wanted to keep as many similarities as possible, but replace differences with wildcards. Expressed this way, the problem reduces to determining a longest common subsequence (LCS) of two strings and inserting wildcards between the fragments which form the subsequence.

We determine the LCS of two strings using a dynamic programming algorithm that is $O(n^2)$ both in time and space. The process uses two nested loops to fill a table with dimensions $(n+1) \times (m+1)$, where n and m are the lengths of the two strings A and B , respectively. For each cell $M_{i,j}$, the i th character of A is compared to the j th character of B , and the cell's value is calculated using the formula

$$M_{i,j} = \begin{cases} \text{if } A_i = B_j, & M_{i-1,j-1} + 1 \\ \text{if } A_i \neq B_j, & \max(M_{i-1,j}, M_{i,j-1}) \end{cases}$$

Whenever two characters match, the value of the cell $M_{i-1,j-1}$ is increased by one; if the characters do not match, then the larger of the two values $M_{i-1,j-1}$ and $M_{i-1,j}$ is chosen. We also store a reference to the cell from which the new value originated. Once the table has been filled in this fashion, the algorithm follows those references, beginning with cell $M_{n,m}$,

until either $i=0$ or $j=0$. A longest common subsequence can be derived by considering the letters on this path that matched. Figure 1 provides an illustration of the LCS algorithm and computes "BDAB" as a longest common subsequence of the strings "ABCBDAB" and "BDCABA".

Instead of returning a longest common subsequence as one string, our implementation returns a list of subsequence fragments that make up the subsequence. The fragments are consecutive character matches and correspond to diagonal stretches of the path marked in Figure 1. A wildcarded URL can then be generated by inserting wildcard characters in between those fragments.

The overall task of generating wildcarded URLs for ad blocking is not this trivial, though. In most cases, more than two URLs will be entered by the user, and while we could create a wildcarded URL by merging all of them together, this would result in a degenerate URL like "http://*" or maybe even just "*", both of which are far too broad. Instead, our algorithm needs to selectively merge two URLs only when the resultant wildcarded URL is sufficiently specific, i.e. they input URLs are similar enough.

A generated URL is specific if there are characters other than wildcards present. However, we do not want to consider substrings that occur very often anyway, such as "http://", "www", top-level domains like ".com" and ".edu", or file format suffixes like ".jpg" and ".gif". Most of the URLs that are compared probably contain at least one of these substrings, so if two URLs match only because they contain these frequent substrings, they still are not sufficiently similar to be merged: The result "http://www*.com*.gif" is not desirable. We also exclude non-alphanumeric characters such as colons, periods, and slashes, as well as the wildcard itself. In determining whether two URLs are similar enough to be merged, we therefore remove occurrences of these substrings and characters from the generated wildcarded URL, and only accept the merge if the result is still non-empty. The example above, for instance, would result in an empty string and thus a rejection of the merge.

Often, a longest common subsequence also includes countless fragments that consist of only a single character. When the two URLs "http://ad.domainname.com/ads/someimage.gif" and "http://

“http://xxx.domainname.com/ads/anotherpic.jpg” are merged, for example, the wildcarded URL generated thus far is “http://*.domainname.com/ads/*o*e*i*.*”. This is a valid, specific merge, but it is neither beautiful nor effective in most cases, since it does not match many other URLs that follow a similar pattern. By removing very short fragments from a longest common subsequence, we can generate the wildcarded URL “http://*.domainname.com/ads/*”, which is simpler and matches more similar URLs. Since a merge that results in only short fragments is likely to be too broad as well, short fragments are also excluded when deciding if two URLs are sufficiently similar.

When we began implementing the generation of wildcarded URLs, we decided to keep these URLs separate from those the user has entered manually; therefore, we added *two* additional lists to AdBlock. The first list contains the original, unmodified URLs of the items the user wants blocked, the second list holds the URLs that are generated automatically. Only the second list is used to actually block ads; there is no need to resort to the original URLs, since the merged URLs necessarily match them as well.

Whenever the user decides to block another URL, it is added to the first list. Then it is compared to the existing wildcarded URLs in the second list to see if a merge is possible. If a candidate for merging is found, it gets replaced by the URL generated by the merge; if no URL in the second list is similar enough, the new URL is added to the second list without modification. Listing 1 provides pseudo-code for the procedures described in the paragraphs above.

Discussion and Improvements

In Figure 2, we present the results of a sample merge: Seven URLs given as input were collapsed into three wildcarded URLs. The first wildcarded URL is a perfectly valid example that matches the quality of a user-entered filter. The third URL demonstrates a case in which the algorithm decided not to merge, since no similar URLs were present in the URL corpus at that time. The second URL that was generated, however, shows a problem of our current algorithm: The sixth and seventh input URL had the common substrings “ollege” and “120”, so the algorithm decided that they can be merged. The result, however, is a wildcarded URL that is very broad and that might block legitimate content as well.

```
CAN-MERGE(A, B) → BOOLEAN
1. FRAGMENTS ← LCS(A, B)
2. S ← empty
3. for each F in FRAGMENTS do
4.   REMOVE-NON-ALPHANUM(F)
5.   REMOVE-COMMON-SUBSTR(F)
6.   if LENGTH(F) < 3 then
7.     F ← empty
8.   end if
9.   S ← APPEND(S, F)
10. end for
11. return not (S = empty)
```

```
MERGE(A, B) → URL
1. FRAGMENTS ← LCS(A, B)
2. S ← empty
3. for each F in FRAGMENTS do
4.   S ← APPEND(S, F)
5.   S ← APPEND(S, ‘*’)
6. end for
7. return S
```

```
ADD-URL-TO-LIST(URL, LIST)
1. for each W in LIST do
2.   if CAN-MERGE(W, URL) then
3.     REMOVE(W, LIST)
4.     M ← MERGE(W, URL)
5.     INSERT(M, LIST)
6.   return
7. end if
8. end for
9. INSERT(URL, LIST)
```

Listing 1: Pseudo-Code for URL Merging

```
ADD-URL-TO-LIST-BEST(URL, LIST)
1. ACCU ← null
2. BEST ← empty
3. for each W in LIST do
4.   if CAN-MERGE(W, URL) then
5.     M ← MERGE(W, URL)
6.     if LENGTH(M) > LENGTH(BEST) then
7.       ACCU ← W
8.       BEST ← M
9.     end if
10.  end if
11. end for
12. if not (ACCU = null)
13.  REMOVE(ACCU, LIST)
14.  INSERT(BEST, LIST)
15. else
16.  INSERT(URL, LIST)
17. end if
```

Listing 2: Pseudo-Code for Best-Case Merging

Input:

```
http://ads.collegehumor.com/ads/quaaffer1.gif
http://ads.collegehumor.com/ads/booble.boobs.120.ch.bounce.gif
http://ads.collegehumor.com/ads/ch120m1c.gif
http://ads.collegehumor.com/ads/_16.120_2.gif
http://www.sportscrew.com/banners/link/ch/college-120.gif
http://ad.doubleclick.net/ad/N635.CollegeHumor/B1476373.2;sz=120x120;ord=673977?
http://www.bullz-eye.com/pictureofday/today120.jpg
```

Output, original algorithm:

```
http://ads.collegehumor.com/ads/*.gif*
http://*ollege*120*
http://www.bullz-eye.com/pictureofday/today120.jpg
```

Figure 2: Sample Merging Results, First-Possible Merge

Input:

```
http://www.sportscrew.com/banners/link/ch/college-120.gif
http://ads.collegehumor.com/ads/nerds.jpg
http://ads.collegehumor.com/ads/8.jpg
http://ads.collegehumor.com/ads/modchip_120x1202.gif
http://www.bullz-eye.com/pictureofday/today120.jpg
http://php.fark.com/pa/adview.php?what=zone:2&n=a16e53ff
http://php.fark.com/pa/adimage.php?filename=1sl_125x250.gif&contenttype=gif
```

Output:

```
http://www.sportscrew.com/banners/link/ch/college-120.gif
*http://ads.collegehumor.com/ads/*
http://www.bullz-eye.com/pictureofday/today120.jpg
*http://php.fark.com/pa/ad*.php?*
```

Figure 3: Sample Merging Results, Best-Case Merge

It is also very likely that the number “120” will disappear after a few more merges, generating a wildcarded URL that rejects anything containing “ollege”.

To prevent the creation of such general wildcarded URLs, we are considering not merging URLs across domains. This is a sensible approach for many ad servers, since the structure of an advertisement URL usually is company- and thus server-specific. We are also thinking about taking more of the directory structure into account, even though this might not prove effective – on many servers, directories are virtual and do not correspond to the actual location of the files. This allows for the same document to be accessible by two different URLs, something our current algorithm can detect and block. Making directories significant might make auto-generation too restrictive.

Our algorithm is also order-dependent; depending on what URL is selected for a merge, the set of blocked

elements is different. From time to time, it may thus be useful to regenerate all the automatic filters from scratch, which is why the first list of unmodified URLs has to be maintained. This feature is available in Adblock’s preferences dialog. Additionally, we currently merge as soon as we find a *possible* merge. We assume it will be better to cycle through all URLs and then execute the *best* merge. We are in the process of testing this new algorithm, outlined as pseudo-code in Listing 2. Figure 3 provides some preliminary results.

This algorithm is still order-dependent, but to a lesser degree: The order of the URLs in the corpus does not matter anymore, but the order in which URLs are added still does. A solution to this might be to store a tree that describes which URLs were merged and whose internal nodes are intermediary URLs that were generated but that are not in use anymore. If combination with such an internal node generates a superior merge, the tree could be restructured appropriately. This approach would require a

completely different data structure than the one AdBlock currently uses and would thus require an extensive rewrite.

We also plan to treat numbers separately. Currently, a number is nothing more than a sequence of digits; therefore, a merged URL might match URLs containing one-digit numbers in a certain place, but not match numbers with more digits. By recognizing numbers as single entity, the merged URLs would block numbers with an arbitrary number of digits, as long as those numbers occur in the correct place.

Despite the problems the algorithm still exhibits in some cases, we believe our “two-click blocking” feature to dramatically lower the difficulty of maintaining an effective blacklist.

4.2. BLACKLIST WEB UPDATES

The second problem we identified in our initial study of AdBlock was the restricted way to share and update a blacklist once it has been created. Even though AdBlock is a program that operates on content downloaded from the world wide web, it does not make use of the internet in a constructive way to enhance its capabilities.

AdBlock only allows the blacklist to be exported to and imported from files on the user’s computer, a solution that does not scale to larger user groups. Since novice users are often also confused by the computer’s file system, we decided to bypass the file system by allowing online updates of the blacklist. This feature was seamlessly integrated into AdBlock’s Tools submenu.

It enables magazines, universities, and company system operators to publish their well-engineered blacklists, which users can then download at the touch of a button. With the built-in possibility for auto-updates, the plugin can periodically update the blacklist from a specified source, relieving users of having to manage their blacklists manually.

Downloading new blacklist filters could be implemented on a domain-by-domain basis as well: Instead of storing all filters available on some central repository, the local computer maintains only a list of the domains the filters affect. Whenever a domain is accessed, this list is consulted to determine if filters are available for the domain, and if that is the case,

new filters are automatically downloaded for this domain only.

We believe this feature is only a first step towards a larger online ad blocking community. With companies investing more money into online advertising, strategies for ad blocking circumvention will be used more often. A community of thousands of users, collectively sharing filters, might be able to counteract those strategies.

4.3. DOM PATH BLOCKING

The use of text advertisement is another tactic advertisers employ to circumvent ad blocking. Since AdBlock can only block ``, `<EMBED>`, `<OBJECT>`, and `<IFRAME>` tags, an advertisement consisting of just text currently cannot be blocked. Most of the time, these ads are still embedded into a recognizable HTML structure: They might be placed in a table, for example, which could be recognized and blocked. We decided to enhance AdBlock by allowing all HTML tags in a document to be blocked. This feature increases the power of AdBlock considerably.

Algorithm and Implementation

An HTML document can be interpreted as a tree with nodes representing HTML tags and edges describing tag nesting. The DOM is an example of such a tree. Since any node in a tree can be uniquely identified by providing a path from the tree’s root to the element in question, we describe an HTML tag that is to be blocked as a DOM path starting at the `<HTML>` root node.

This path is expressed as a string of the form `Tag:Index(/Tag:Index)*`, where `Tag` describes the kind of tag and `Index` selects the instance of the tag that should be entered, beginning with 1. The path `body:1/table:2/tr:3`, for example, identifies the third row in the second table within the document’s first body. Most of the time, the element to be blocked is specific to only a few websites. We therefore combine the path with a wildcarded URL as used elsewhere in the project. Since both the path and the URL are strings, they can easily be joined. To distinguish path-URL pairs from regular URLs, we decided to prefix them with “D|”; a full path-URL string looks like “D|body:1/table:2/tr:3|http://www.somedomain.com/*”.

GET-PATHS(URL, PATH-URL-PAIRS) → PATH-LIST

```
1. PATHS = empty
2. for each (P, U) in PATH-URL-PAIRS do
3.   if REGEXP-MATCH(URL, U) then
4.     INSERT(P, PATHS)
5.   end if
6. end for
7. return PATHS
```

BLOCK-PATH(DOM-NODE, PATH)

```
1. if PATH = empty
2.   HIDE(DOM-NODE)
3.   return
4. end if
5. (TAG, INDEX) ← FIRST(PATH)
6. COUNT ← 0
7. SUBNODE ← null
8. for each CHILD in DOM-NODE do
9.   if TYPE(CHILD) = TAG then
10.    COUNT ← COUNT + 1
11.    if COUNT = INDEX then
12.     SUBNODE ← CHILD
13.     break
14.    end if
15.   end if
16. end for
17. if COUNT = INDEX then
18.  REMAINDER ← REST(PATH)
19.  BLOCK-PATH(SUBNODE, REMAINDER)
20. else
21.  error
22. end if
```

BLOCK-ALL-PATHS(URL, PATH-URL-PAIRS)

```
1. DOM-ROOT ← GET-DOM-ROOT(URL)
2. PATHS ← GET-PATHS(URL, PATH-URL-PAIRS)
3. for each P in PATHS do
4.   BLOCK-PATH(DOM-ROOT, P)
5. end for
```

Listing 3: Pseudo-Code for DOM Blocking

We had hoped to integrate this into the existing blocking mechanism. Unfortunately, the content management policy the Mozilla platform employs allows custom treatment only for those elements the original Adblock already blocked. For reasons unknown to us, Mozilla treats only images, plugins, and frames as configurable content; therefore, arbitrary tags cannot be blocked using a policy.

As an alternative, we implemented an `EndDocumentLoad` observer that Mozilla activates whenever a page has been loaded completely. This observer then matches the document's URL against the URL portions of all stored path-URL pairs. If a match is found, the path associated with that URL is

added to a list, and all elements contained in this list are blocked. If no match is found, the website is displayed without modification.

The blocking algorithm considers each path in the list individually and recursively enters the DOM until the node to remove is reached. The node is removed by setting its display style to “none” or “hidden”, causing it to disappear. If the path specifies an element that does not exist, the traversal for this particular path is aborted. Listing 3 provides pseudo-code for determining the paths to be blocked and hiding the elements they identify.

Discussion and Improvements

The current implementation of DOM path blocking works correctly, but it performs unsatisfactorily when web pages load slowly: The user can watch as blocked content gets downloaded and displayed; the items are hidden only after the entire page has been loaded. For the same reason, we currently cannot prevent an unwanted element from being loaded. The best solution, of course, would be to generalize Mozilla's content management policy to apply to arbitrary elements. This would require a change to the Mozilla platform and specification, though.

If filtering were to occur *while* as opposed to *after* content is loaded, items could be removed as soon as they are added to the DOM. We investigated this option and implemented a `StartDocumentLoad` observer, which then installs event handlers to be notified on DOM insertions. Unfortunately, the necessary DOM mutation events, namely `DOMNodeInsertedIntoDocument` or `DOMSubtreeModified`, have not yet been implemented or behave incorrectly, as several bug reports on Bugzilla document (see [8], for instance). Since the `DOMNodeInserted` event has been implemented, the behavior of `DOMNodeInsertedIntoDocument` could be achieved by attaching event handlers to all the nodes that get created, i.e. the event handler for the root node attaches new event handlers to children of the root node once they get inserted, which in turn attach new event handlers to the children of the nodes they are attached to, and so on. This requires one JavaScript event handler per node, though, which would probably make the process rather costly.

We also plan to expand the expressiveness of the paths by including wildcards in place of indices or arbitrary path segments. The “#” wildcard, for

example, would declare that any index should be blocked, instead of just a particular one specified by a number. The path “body:1/table:#/tr:2” would block the second row of all tables contained directly within the document’s body. The “*” wildcard could be used as a placeholder for arbitrary path segments. Using that scheme, the path “*/tr:2” would block the second row of all tables, wherever they occur. The recursion model in Listing 3 would have to be changed for this, though, since wildcarded paths can identify multiple elements in different branches of the tree. Currently, the algorithm recurs into one and only one branch.

As proof-of-concept we devised DOM paths that block advertisements on two popular websites: Google.com’s search text ads and Fark.com’s advertising bar. These paths, as well as a few other example adblocking paths, can be imported from the URL <http://www.xiphoris.com/adblock.txt>.

We have begun to implement automatic path generation according to a context menu, as provider for automatic wildcard generation. Currently, any web page element can be selected and automatically DOM blocked, which adds its DOM path and the current URL to the DOM path blacklist. Future improvements include streamlining this process, e.g., the current selection could be highlighted with a blinking frame. Though DOM path blocking will remain a powerful but more difficult feature, path selection using the context menu should put it within reach of most users.

5. CONCLUSION

With larger and more intrusive online advertisement on the horizon, the importance of tools like Adblock will indubitably increase in the future. By providing a simple “two-click blocking” feature and allowing blacklists to be shared online, we enable novice users to effectively block online ads. Furthermore, we are certain that DOM path will be a cornerstone in blocking many different kinds of advertisements, including previously unblockable text ads.

The project is currently available at <http://www.owl.net.rice.edu/~mgricken/527/>. In the future, we will seek to incorporate our improvements into the general Adblock project.

REFERENCES

- [1] [The Internet Coaching Library](#). Internet World Stats. Updated 24 Oct. 2004. The Internet Coaching Library. Accessed 5 Dec. 2004 <<http://www.internetworldstats.com/stats.htm>>
- [2] [ITFacts.biz](#). Advertising category. ITFacts.biz. Accessed 5 Dec. 2004 <<http://www.itfacts.biz/index.php?id=P2012>>
- [3] [ScreenPlays](#). New Web Video Ad and Metrics Tools Portend Surge in Internet Ad Spending. Updated 19 Aug. 2004. ScreenPlays. Accessed 5 Dec. 2004 <<http://www.screenplays.bz/sp704g.html>>
- [4] [The Adblock Project](#). Home page. The Adblock Project. Accessed 5 Dec 2004 <<http://adblock.mozdev.org/>>
- [5] [The Mozilla Organization](#). Home of the Firefox web browser. Updated 4 Dec. 2004. The Mozilla Organization. Accessed 5 Dec. 2004 <<http://www.mozilla.org/>>
- [6] [World Wide Web Consortium \(W3C\)](#). Document Object Model (DOM). Updated 7 Apr. 2004. W3C. Accessed 6 Dec. 2004 <<http://www.w3.org/DOM/>>
- [7] [The AdblockLearner Project](#). AdblockLearner Project Page. The AdblockLearner Project. Accessed 6 Dec 2004 <<http://adblocklearner.mozdev.org/>>
- [8] [The Mozilla Organization](#). Bugzilla Bug 74219. Updated 10 May 2004. The Mozilla Organization. Accessed 6 Dec 2004 <https://bugzilla.mozilla.org/show_bug.cgi?id=74219>

ACKNOWLEDGEMENTS

We thank the Mozilla Organization, the Adblock Project, Dr. Dan Wallach, Scott Crosby, and COMP 527 at Rice University for supporting us during this project.