Detecting malicious activities with user-agent-based profiles

INTERNATIONAL JOURNAL OF NETWORK MANAGEMENT

Int. J. Network Mgmt 2015; **00**:1–25

Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/nem

Detecting Malicious Activities with User-Agent Based Profiles

Yang Zhang[1], Hesham Mekky[1], Zhi-Li Zhang[1*] Ruben Torres[2], Sung-Ju Lee[4], Alok Tongaonkar[2] and Marco Mellia[3]

[1]*University of Minnesota*

[2]*Symantec Corporation*

[3]*Politecnico di Torino*

[4]*Korean Advanced Institute of Science and Tecnology*

SUMMARY

Malicious activities have become a primary security threat after hosts are infected. Attackers typically use HTTP to carry out malicious activities, such as botnets, click fraud and phishing, as they can easily hide among the large amount of benign HTTP traffic. The User-Agent (UA) field in the HTTP header carries information on the application, OS, device, etc., and adversaries fake UA strings as a way to evade detection. Motivated by this, we propose a novel *grammar-guided* UA string classification method in HTTP flows. We leverage the fact that a number of "standard" applications, such as web browsers and iOS mobile apps, have *well-defined* syntaxes that can be specified using context-free grammars, and we extract OS, device and other relevant information from them. We develop *association* heuristics to classify UA strings that are generated by "non-standard" applications that do not contain OS or device information. We provide case studies that demonstrate how our approach can be used to identify malicious applications that generate fake UA strings to engage in fraudulent activities.

Received . . .

KEY WORDS: Network Security; HTTP User-Agent; Flow Correlation

*Correspondence to: 4-192 Keller Hall, 200 Union Street SE, Minneapolis, MN 55416. E-mail: zhzhang@cs.umn.edu

1. INTRODUCTION

HTTP has become the *de facto* application-layer "transport" protocol, over which many applications such as JSON, SOAP, gaming, VoIP, video streaming, and software updates operate. From the perspectives of network measurement and traffic analysis, it is important to be able to classify and separate various applications transported over HTTP. Such capability is particularly useful in aiding network security tasks such as malware detection, mainly because HTTP has become the main medium for illicit activities on the Internet such as drive-by downloads [18], phishing [19], botnet command-and-control (C&C) [20], click frauds [24], and so forth.

Given a collection of HTTP flow traces (i.e., network traffic over TCP port 80) passively captured within a network where HTTP and TCP/IP header information is collected, we are interested in developing an effective and robust method to classify and separate various applications transported over HTTP. To aid network security monitoring, our emphasis is on identifying *anomalous* applications such as "handcrafted" web clients that mimic "standard" browsers or malicious applications that conduct fraudulent activities such as click frauds. To this end, we want to robustly separate HTTP flows generated by normal, benign applications such as legitimate web browsers (e.g., Internet Explorer, Firefox, Chrome) and other commonly used applications (e.g., iOS or Android mobile apps) from "anomalous" applications.

One key feature we focus on is the `User-Agent` (UA) HTTP header field, which is sent by a web browser to a web server to convey the client's operating system, browser type and version, the rendering engine, and the application name in the case of traffic from mobile devices [25]. Web servers utilize such information to customize their response to the web browser for proper rendering. However, the UA string is also used by malware for illicit activities, for instance, as a way to spoof a legitimate browser being used by a client on click fraud events, as a way to leak personal information from the infected host, or to communicate with the Command-and-Control (C&C) server [12]. More recently, the UA string has been used as a way to exploit servers vulnerable to the Shellshock [11] attack.

Prepared using nemauth.cls
DOI: 10.1002/nem

The big challenge for network administrators and security analysts is that it is difficult to differentiate between legitimate and malicious UA strings. The reasons are two-fold. First, although there is a standard UA format defined in RFC 7231 [8], not all benign applications follow it (e.g. Table V), which limits the possibility of filtering HTTP connections with non-standard UA strings. Second, if a malicious UA string is following the standard format, there is no obvious way to detect it, except for point solutions as we describe in related work.

Due to the diversity of UA strings, the state-of-the-art mechanisms for processing UA strings utilize a set of (*ad hoc*) pattern matching heuristics based on regular expression (regex) rules (see, e.g., [7]). These tools are designed for web servers to recognize browsers for content rendering, *not* for separating normal browsers from anomalous ones. They are generally ineffective in recognizing *non* browser applications that often include partial information (e.g., only the name of an application but no OS or device type) in an application-specific format or even a random-looking character string (see Section 5 for examples).

In this paper, we present a novel approach to robustly separate and classify applications transported over HTTP using the UA strings, with the goal of detecting malicious applications or activities. This approach consists of two components. 1) We develop a novel context-free grammar (CFG)-based method for efficiently and robustly parsing the UA strings generated by "standard" applications such as common web browsers, iOS and Android apps. 2) To cope with "non-standard" applications with (possibly arbitrarily formatted) UA strings that contain only partial or little information about the application (e.g., OS-type), we leverage the associations between the UA strings and hostnames contained in the HTTP flows to classify "non-standard" applications which generate these UA strings. For example, although many anti-virus (AV) engines often generate HTTP flows with "random-looking" UA strings, we observed that these flows are always directed to well known AV company websites (e.g., `kaspersky.com`)). We combine these two components to build profiles of various known applications based on information extracted from UA strings. Then, deviations of these profiles can be used as indicators of potentially anomalous activities, generated by "malicious" applications running at an infected device.

Prepared using nemauth.cls
DOI: 10.1002/nem

We use a 24-hour dataset from a large, nation-wide ISP, collected in April 2012, to evaluate our methodology. The monitored network is mostly residential, with high-speed ADSL connections to the Internet. The collected data includes all inbound/outbound TCP connections to the network. The dataset contains only the TCP and HTTP header information (the HTTP payload was not analyzed and the IP addresses were anonymized to preserve privacy). Our dataset includes over 40 million HTTP connections from over $15,000$ unique client IP addresses. After the data collection, malicious flows were labeled by a commercial IDS.

In summary, our contributions are three-fold: *(i)* We develop a novel *CFG-based parser* for classifying UA strings generated by "standard" applications over HTTP that is modular and more effective than the state-of-art regex-based tools (see Section 4). *(ii)* We develop a novel UA string-hostname association method for classifying UA strings generated by many "non-standard" applications (see Section 5). *(iii)* We incorporate these mechanisms into a *proof-of-concept* system (see Section 3 for an overview of the overall methodology and system) which builds application profiles based on information extracted from the UA strings and employs several inference mechanisms to identify anomalous UA strings/applications. We tested our system in a 24-hour network trace from a nation-wide Internet Service Provider (ISP), and present a number of case studies to illustrate how various attacks with different artifacts such as non-standard UA strings, and "fake" UA strings that mimic standard cases can be detected (see Section 6).

2. RELATED WORK

There has been a variety of heuristics and tools proposed for classifying UA strings, most of which are developed to help web servers identify the client browsers so as to provide appropriate web content. Many of these methods simply rely on building and maintaining a database of various UA strings seen in the wild (e.g., browscap.ini and borwscap.dll used by Windows web servers [6]). Others combine such methods with *regular expression* based classification rules [7, 13], where a laundry list of rules is supplied by individuals and accumulated over time. Our experience in using

Prepared using nemauth.cls
DOI: 10.1002/nem

these tools reveals that these tools often produce incorrect classification results. This is due to the complexity of the rules and the large variation of the UA string formats. In addition, these rules become difficult to debug and manage as they grow in size.

This motivated us to try MAUL [21], which is a machine learning (ML) based UA classification scheme. The problem with ML based schemes is its high false classification rates, as it cannot distinguish slight differences between valid and invalid UA strings. For instance, invalid UA strings such as “Mozilla Mozilla Mozilla” or “Chrome Mozilla Windows” will be classified as browsers. In contrast, context-free grammar based classification scheme not only classifies standard UA strings more accurately, but also detects syntactic errors and other anomalies in browser-like UA strings.

Finally, UA strings have also been utilized to detect malicious activities, e.g., for detecting SQL injection attack [14]. In particular, Kheir [23] finds that anomalous UA strings are often associated with malware activities. We apply our grammar-based UA parsing method to show how they can be systematically utilized to detect not only malicious applications with unique “strange-looking” UA strings, but also those that attempt to mimic normal applications.

3. WALKTHROUGH

In this section, we walkthrough the paper, which includes the motivation for our work and an overview of our methodology and proof-of-concept system design.

3.1. Motivation for our work

Initially, we noticed that existing methods to classify UA strings were used mainly by web servers to improve page rendering, and they were solely based on regular expressions (regexes). The regexes are organized in a hierarchy and there is a parsing library that will iterate over them. For a given UA string, the parsing library will try to match a regex in the first level of the hierarchy, which tries to match a browser name (e.g. Firefox or Chrome). Then, the library will iterate over all regexes in all subsequent levels until one rule matches completely. In our experiments, we found that these methods are hard to extend, and are extremely slow for network monitoring environments.

Prepared using nemauth.cls
DOI: 10.1002/nem

For example, a web server may want to inspect 100 UA strings per second, but an ISP middlebox monitoring the network may see millions of UA strings per second. Therefore, the iterative matching over regexes is not scalable or suitable for ISPs. Fortunately, regexes plus the library composed of *if/else* statements can be easily expressed using a Context Free Grammar (CFG). In addition, CFGs are easy to extend, they walk the grammar tree quickly, and are more efficient than iterating over regexes. Consequently, we chose to build our prototype using a CFG engine that identifies standard UA strings.

3.2. Methodology

An accurate CFG engine is critical to identify standard UA strings. Hence, we propose a semi-manual process to write the CFG engine. This process is composed of three phases: (1) extract UA strings from sandbox environments, (2) extract UA strings from network traces, and (3) write the CFG using standard UA strings from phases one and two. In phase one, a sandbox environment can be used to run different versions of popular web browsers, and their popular plugins. Then, the generated UA strings are added to a repository with a metadata string, which describes the UA strings in the extracted group (e.g. UA strings for Firefox running on Windows with Firebug plugin are added to the repository with metadata "Firefox, Windows, Firebug"). In phase two, we use UA strings extracted from HTTP requests in real ISP network traces and we manually extract the standard UA strings. We speed up the manual process by using regexes. For instance, we extract all UA strings containing "Firefox", then we manually reject all the non-standard ones, and add the standard UA strings to the repository with their corresponding metadata. In phase three, we use the standard UA strings in the repository to incrementally write the CFG (e.g. we write CFG for Firefox UA strings, then we add another group of UA strings until we get the complete CFG). We use this CFG engine in our prototype to identify standard UA strings.

Unfortunately, non-standard UA strings lack structure and cannot be identified using CFGs. However, after manually analyzing our ISP network traces, we found that some non-standard UA strings possess characteristics that can be used to develop heuristics to group them together. For

Prepared using nemauth.cls
DOI: 10.1002/nem

instance, the majority of benign non-standard UA strings in our network traces belong to AV and software updates. These UA strings are highly random and a UA string shows up only once since it includes a cryptographic hash of information that AV is sending to the remote server. However AV related UA strings are always associated to a AV company domain name, such as `symantec.com`. We use such characteristics to develop a detection engine for non-standard UA strings.

3.3. System Overview

Figure 1 illustrates a proof of concept for our system. The input to the CFG engine is the HTTP flows per client IP, and the output is the standard UA strings and the non-standard UA strings. First, the standard UA strings are used to create a profile for each application instance running on a single machine, where this profile includes browsers (types and versions), OS (types and versions), devices (e.g, Mac, iPad, PC, etc.), and applications (e.g., mobile apps). These profiles are used later by the inference engines to find anomalies and suspicious activities. Second, the non-standard UA strings are fed into the *Flow Grouper*, where UA strings are grouped based on top two level domain names, e.g., all UA strings with domain names `symantec.com` are grouped together. Then, *Hostname Association* uses a set of heuristics to label UA strings that poses similar characteristics (see Section 5). Some UA strings might still be unknown, and therefore we fall back to other source of information like googling them to process these UA strings. Then, the inference engines are used to search for conflicting or anomalous UA strings (see Section 6). Finally, the system produces a report of benign and malicious activities.

4. DESIGNING APPLICATION PROFILES FOR STANDARD UA STRINGS

We describe our approach to parse standard UA strings and extract key information from them, such as browser type and operating system. We begin by describing regex-based UA string parsers, a technique typically used in the industry today, and its limitations. Next, we present our UA string parser, which consists of a series of per-application context free grammars.

Prepared using nemauth.cls
DOI: 10.1002/nem

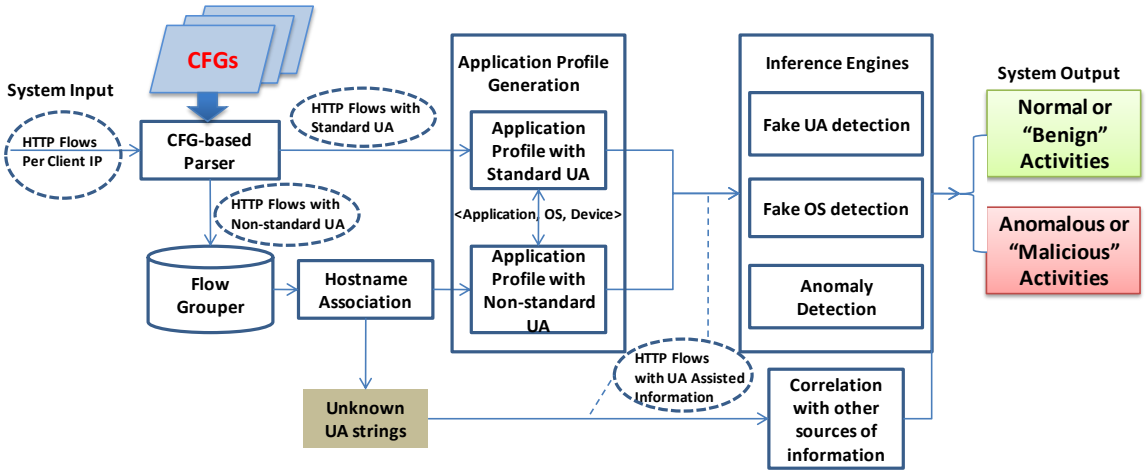

Figure 1. System Architecture

4.1. Parsing UA Strings with Regular Expressions

A straightforward solution to this problem is to build a huge list of regexes that represents all possible UA strings in the Internet today. In such a system, an incoming UA string is matched against the list of regexes and the first one that matches is used to extract any key information from the string. BrowserScope [7] is an example of this approach. However, this method has several problems:

An inappropriate regex matching order can cause false positives. Short regexes are typically less strict than long regexes and tend to match more often. This can cause false positives. For instance, consider the following UA string: `Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.3) Gecko/20060426 Firefox/1.5.0.3 (.NET CLR 3.5.30729) GoogleToolbarFF 3.0.20070525`. This is the UA string of Firefox using the GoogleToolbar extension. However, if shorter regexes that match Firefox flows are tested first, the application is mislabeled as Firefox, instead of GoogleToolbar.

A linear scan over many regexes is required and degrades performance. In most cases, a single standard UA string will be matched against multiple regexes before a match is found, since there is no optimization in the regexes to match. Furthermore, in the case of non-standard UA strings, all the regexes in the long list might have to be matched. These cases might cause performance degradation. A possible solution is to build different lists of regexes that serve different purposes e.g., one regex list to match possible devices, and another regex list to match possible operating

Prepared using nemauth.cls
DOI: 10.1002/nem

systems, and so on. This method could reduce the total number of regexes to scan, but we still need to linearly scan each individual list to find different components. In addition, a potential problem is that because there are dependencies between device, OS and application, which are ignored in this approach, some fraudulent matches might be considered as valid standard UA strings. In Section 6, we describe an approach to solve this issue.

Some UA strings cannot be expressed by regular expressions. In our dataset, we found the case where web browsers are embedded in third party applications, which generate nested UA strings that look like the one below: `Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; GTB7.3; Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1) ; (R1 1.6); .NET CLR 1.1.4322; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; msn OptimizedIE8;ITIT)`. These nested UA strings cannot be expressed using standard regexes, because they are not regular [22].

Updating heuristic rules is not flexible. Updating heuristic rules in regular expression based system is an important operation if the system were to be widely deployed in practice. Clearly, such a system need to be updated as new versions of the application are released. However, the core of such a system is a laundry list of regular expressions. It is not flexible to make it updated because the newly added rules can confuse existing regex-based method. For instance, new versions of Opera (Opera 15+) have adopted a different format, where the old formats always contain the keyword "Opera" (either at the beginning or the end), but the new ones end with "OPR/[version]" [15].

To alleviate all these limitations above, we build per-application context-free grammar parsers for UA strings, which we describe in next subsection.

4.2. Parsing UA Strings with BNF-based Context Free Grammars

To parse UA strings, we first identify the UA strings generated by popular applications such as commonly used web browsers and iOS-based apps which have a well-defined syntax. We noticed that the syntaxes for the UA strings of these applications can be best specified using *context-free grammars* (CFG) in terms of Backus-Naur Form (BNF). Using existing compiler tools, we developed a baseline BNF-based UA string parser to recognize those that are generated by these

Prepared using nemauth.cls
DOI: 10.1002/nem

Table I. Example User-Agent strings generated by popular browser types.

MSIE	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 6.1; en-US; .NET CLR 1.1.22315)
	Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; SV1; .NET CLR 2.0.50727)
	Mozilla/4.0 (compatible; MSIE 9.0.8112.16443; Windows NT 6.1)
	Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; Trident/6.0)
Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:29.0) Gecko/20120101 Firefox/29.0
	Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.0.7) Gecko/2009021906 Firefox/3.0.7
	Mozilla/5.0 (X11; U; SunOS i86pc; en-US; rv:1.9.0.6) Gecko/1986081808 Firefox/3.0.6
	Mozilla/5.0 (X11; U; Linux i686; de-DE; rv:1.7.6) Gecko/20050306 Firefox/1.0.1
Chrome	Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/37.0.2049.0 Safari/537.36
	Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_6; en-US) AppleWebKit/530.5 (KHTML, like Gecko) Chrome/ Safari/530.5
	Mozilla/5.0 (Linux; U; en-US) AppleWebKit/525.13 (KHTML, like Gecko) Chrome/0.2.149.27 Safari/525.13
	Mozilla/5.0 (Linux; Android 4.0.3; GT-I9100 Build/IML74K) AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.133 Mobile Safari/535.19
Safari	Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_7; en-us) AppleWebKit/534.16+ (KHTML, like Gecko) Version/5.0.3 Safari/533.19.4
	Mozilla/5.0 (iPad; CPU OS 6_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/6.0 Mobile/10A5355d Safari/8536.25
	Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-us) AppleWebKit/418.9 (KHTML, like Gecko) Safari/419.3
	Mozilla/5.0 (Windows; U; Windows NT 5.1; it) AppleWebKit/522.13.1 (KHTML, like Gecko) Version/3.0.2 Safari/522.13.1
Opera	Opera/9.80 (X11; Linux x86_64; U; en) Presto/2.9.168 Version/11.50
	Opera/12.80 (Windows NT 5.1; U; zh-cn) Presto/2.10.289 Version/12.02
	Opera/9.80 (X11; SunOS sun4u; U; en) Presto/2.2 Version/10.11
	Opera/9.80 (Macintosh; Intel Mac OS X 10.6.8; U; en) Presto/2.9.168 Version/11.52 Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.0) Opera 12.14

standard applications and extract the type of application, OS version and device information. In the following, we present our approach using popular web browsers as primary examples and discuss the advantages of our BNF-based compiler approach.

Web browsers are perhaps the most popular application used on desktop and laptop machines. It is no surprise that they comprise the majority of UA strings we see in our dataset. Table I shows some examples of UA strings for common browsers found in our dataset. We see that the UA strings generated by these browsers contain similar keywords and share certain structural components. For example, the UA strings generated by IE browsers starts with the keywords “Mozilla/4.0” or

Prepared using nemauth.cls
DOI: 10.1002/nem

"Mozilla/5.0", followed only by a set of keywords enclosed by parentheses, including the "MSIE" term and versions, Windows OS related information and the rendering engine "Trident/[version]" (if present). However, the UA strings generated by Firefox, Chrome and Safari also begin with the same keywords (most commonly with "Mozilla/5.0"), followed by a set of OS-related keywords enclosed by "(...)", and ended with a set of specific keywords starting with the rendering and layout engines "Gecko" or "Applekit" and containing the browser type (e.g., Firefox, Chrome or Safari). On the other hand, the UA strings for Opera browsers begin with "Opera/[version]" (except for newer versions that also begin with "Mozilla/5.0").

In the examples above, all standard browser UA strings manifest (nested) matching structures that are characteristic of context-free languages, e.g., the prefix element "Mozilla/[4.0—5.0]" matches a *rendering engine-browser type* suffix element (or an empty string in the case of IE browsers), and the left parenthesis "(" matches with the right parenthesis ")". Furthermore, the rendering engine-browser type suffix element also contains a matching structure, e.g., "Gecko/[version]" matches only "Firefox/[version]" or "Opera/[version] and "Applekit/[version]" matches only with "Safari/[version]". These (nested) matching structures can be best recognized by push-down automata, i.e., CFGs.

We define a set of CFG production rules using the BNF forms with non-terminal terms and terminal terms (tokens). Some examples are shown in Table II, where the terms in angular brackets $<\ldots>$ indicate *non-terminal* terms, and all other terms that never appear in the left of the production rules (e.g., various symbols and strings inside quotation marks) are *terminal* terms (i.e., "tokens"). In the above examples, note that "" denotes an empty string, while " " denotes a white space, and "|..." indicates that user-specific rules can be added there.

Leveraging existing compiler tools *Flex* [2] and *Bison* [1], we developed a parser for UA strings, in order to classify and extract the browser type, OS, device and other relevant information if it exists. The parser consists of two main components – a *lexical analyzer* and a *syntax analyzer* – and operates in two phases: (i) the lexical analyzer first tokenizes a UA string and extracts each meaningful element (i.e., the terminal terms); and (ii) the syntax analyzer applies the context-free

Prepared using nemauth.cls
DOI: 10.1002/nem

Table II. Example production rules for browser UA strings.

```
<standard-browser>::=<browser-prefix>"("<OS-system>")"<browser-suffix>;
<browser-prefix>::="Mozilla/4.0"|"Mozilla/5.0"|"Opera/"<ver-1-dot>;
<browser-suffix>::=""|<host-info> <render-engine> <opera-version>|
   <render-engine> <browser-type>|<browser-type>|...;
<host-info>::="(<OS-version> <security-level> <language>)"|
              "(<OS-version> <security-level>)"|
              "(<OS-version> <language>)"|
              "(<OS-version>)"|...;
<render-engine>::="Presto/"<ver-two-dot>|
                   "Gecko/"<ver-no-dot>|
                   "AppleWebKit/"<ver-1-dot>|...;
<security-level>::="U"|"I"|...;
<language>::="en"|"es-ES"|"zh-cn"|"zh-tw"|"pl"|"cs"|...;
<OS-system>::="compatible;"<IE><window><IE-suffix>|<window>|<OSX>|...;
<window>::="Window NT "<version>|<additional-window-info>;
<additional-window-info>::="<.net>"|"<plugin>"|...
<.net>::=<.net>|".NET"<ver-1-dot>"C"|".NET"<ver-1-dot>"E"|".NET CLR "<ver-2-dot>;
<plugin>::="Media Center PC "<ver-1-dot>|...;
<IE>::="MSIE "<ver-1-more-dot>|...;
<browser-type>::="Firefox/"<ver-1-more-dot>|"Opera/"<ver-1-dot>|<chrome>|...;
<chrome-safari>::=<chrome-browser>" "<safari-type>|<safari-suffix>|...;
<chrome-browser>::="Chrome/"<ver-1-more-dot>" "<safari>|...;
<safari>::="Safari/"<ver-1-more-dot>|<mobile>"Safari/"<ver-1-more-dot>|...;
<mobile>::="Mobile"|"Mobile/"<alphanumeric-version-no-dot>"|...;
<opera-version>::="Version/"<ver-1-dot>;
<version>::=<ver-no-dot>|<ver-1-more-dot>;
<ver-1-more-dot>::=<ver-1-dot>|<ver-2-dot>|...;
<ver-no-dot>::=<digits>;
<ver-1-dot>::=<digits>"."<digits>;
<ver-2-dot>::=<digits>"."<ver-1-dot>;
<digits>::=<digit><digits>:
<digit>::="0"|"1"|...|"9";
```

BNF production rules to recognize the structure of a UA string that follows the rules and outputs the browser type and other relevant information thus extracted, or otherwise rejects it together with error messages indicating where the syntactic errors occur.

The advantages of context-free BNF-based parser approach for classifying (well-defined) UA strings are the following: (i) it makes the parser scalable and extensible; when new types or versions of browsers are created, we can simply add new production rules or version numbers in the existing rules; (ii) in contrast to a UA parser relying purely on complex regular expression-based heuristics (e.g., [7]), the resulting production rules are more modular and flexible for a human operator to understand and manage; (iii) the error messages generated by the parser provide hints as to how a UA string deviates from the expected standard UA strings, and can be utilized to detect *anomalies*; and (iv) importantly, similar to "type checking" and other runtime techniques used for program verification, we can plug in *browser verification* modules that incorporate "semantic" information to check the validity of the UA strings that have passed the syntax parser. Such semantic constraints

Prepared using nemauth.cls
DOI: 10.1002/nem

can be verified at the last step by invoking appropriate browser verification modules based on the browser type, OS and other relevant information extracted from the UA string that has passed the syntax analyzer. In Section 6 we discuss how we exploit these last two features (iii) and (iv) to help detect and identify "fake" browser UA strings generated by malicious applications.

The UA strings generated by standard iOS (and MacOS) applications also follow a well-defined syntax: <app-name>/<version> CFNetwork/<version> Darwin/<version>. We have defined production rules and developed a parser for parsing the UA strings generated by the standard iOS/MacOS apps. For the UA strings that pass the grammar checking, the distinction between iOS and MacOS is determined by the CFNetwork version number. Other "well-known" applications such as Window Media Center, Media Player, Window Live, standard browser plug-ins, and standard Android apps also follow well-defined syntaxes, and we have developed BNF-based parsers for them.

4.3. Evaluation

To evaluate the effectiveness and efficiency of our CFG-based parser, we compare it with a regex-based UA string parser [7], the current state-of-the-art. UA strings are randomly chosen from our dataset and parsed by both approaches. Then, the running time of each approach is recorded, as shown in Table III. For parsing the same amount of UA strings, CFG-based parser are much more efficient than regex-based parser. This is because [7] requires a linear scan over many regexes, which degrades performance (see Section 4.1 for a detailed explanation).

Table III. UA String Parsing time (in seconds) for our CFGs and Regexes

# Strings	1	2	5	10	50	100	1,000	10,000	100,000
CFG-based parser	0.001	0.001	0.001	0.001	0.001	0.002	0.005	0.042	0.393
Regex-based parser	0.323	0.324	0.328	0.332	0.343	0.359	0.709	4.097	38.211

4.4. Dataset Analysis.

In our dataset we found more than 40 million HTTP flows and $94,876$ *unique* UA strings. Applying our BNF-based UA string parsers for standard applications, we separated these unique UA strings

Prepared using nemauth.cls
DOI: 10.1002/nem

Table IV. Standard browser and non-browser UA string classifications.

Category	**Browser-type**							
	IE	Chrome	Firefox	Opera	Safari	Mobile Browser	Other	Total
UAs	18,527	673	1,255	144	947	827	385	23,298
Flows	9,500,779	8,805,982	7,716,747	163,829	172,979	4,512,137	414,961	31,287,414
Category	**Non-Browser**							
	iOS app	Android app	Other	Total				
UAs	7,425	871	667	8,963				
Flows	1,075,071	56,910	67,244	1,199,225				

into two categories: *standard* UA strings ($32,261$ unique UA strings, about 34%), which matched one of the BNF parsers (i.e., follow well-defined syntaxes), and *non-standard* UA strings ($62,615$ unique strings, about 66%) that did not match any of the BNF parsers. As shown in Table IV, of the standard UA strings, $23,298$ (24.6%) of them matched parsers for browsers and $8,963$ (9.4%) matched *non-browser* parsers for iOS/MacOS, Android and other applications with well-defined syntaxes.

5. HANDLING NON-STANDARD UA STRINGS

In Section 4, BNF grammar rules assign labels to user-agents based on their lexical structure. However, not all UA strings follow the BNF grammar rules. Table V shows examples of this type of UA strings. For instance, AV signature updates and OS updates lack structure and contain random strings. To assign labels to user-agents in this category, we developed a heuristic that we call hostname-based association.

We analyzed our dataset and noticed that many of the non-standard UA strings belong to specific applications, such as AV software. AV software embeds local information (e.g. software version, signatures database version, etc) in the UA string when checking for signature updates. Those UA strings differ from browser UA strings in two aspects: (i) each UA string is unique as it includes a SHA or MD5 hash of the information sent to the server; and (ii) the associated hostname to the UA string belongs to one or two unique top-two level domain names e.g., all flows in this category communicate with a specific AV such as `kaspersky.com` or `kaspersky.net`. Operating system updates (e.g. for Windows and Mac), exhibit similar properties, but they are less random

Prepared using nemauth.cls
DOI: 10.1002/nem

as they do not include any hash of the data. Our heuristic, which we describe below, tries to cover both the AV and the OS update cases.

The key intuitions/ideas behind hostname association method is based on our analysis of the UA strings in our dataset. We find that although there are a significant portion of "non-standard" UA strings, they roughly fall into two categories: 1) UA strings containing mostly fairly random alphanumeric characters; and 2) UA strings containing certain fixed keywords and some loose defined structures. However, a key observation we have made is that the HOSTNAME field in the HTTP flows containing these UA strings provide important hints regarding what type of applications may have generated these UA strings. For instance, for the UA strings in category 1) above, there is often a many-to-one mapping between the UA strings and the (top-2 level) domain name, and the domain name is, say, mcafee.com or kaspersky.com, suggesting that these UA strings – despite they are almost completely random-looking are generated by antivrus sotfware. Similarly, for many UA strings in Category 2 are also a many-to-one mapping or a m-to-n mapping (where m is much larger than n, and n is a smaller number, say, n=2, 3). Even when in some cases where n in the m-to-n mapping is relatively larger, there are a few dominant hostnames (e.g., megaupload.com) which reveal what the applications are, or there are certain patterns (e.g., keywords such as "tracker" or "upload") in the hostnames that echo the keywords in the UA strings (e.g., torrent, BTclient). Our UA string-hostname association method basically applies these observations to heuristically label and classify the non-standard applications (which fall into categories such as antivirus, software update, p2p, media player, etc.). Hopefully this helps explain how the hostname association method works.

To determine whether a UA string is "random" or not, we compute the entropy of the string. In our testing, we have varied the entropy value from 2 bits/byte to 6 bits/byte and find that 4 bits/byte yields the best result.

The flow grouper is used in our system to group all flows with the same top-level hostname together so that we can generate the UA strings to hostname mappings. In terms of how much time you retain the string in the "flow grouper", for simplicity, we actually used the entire 24 hours — we

Prepared using nemauth.cls
DOI: 10.1002/nem

Table V. Example UA strings generated by “non-standard” applications.

AV	*BIXBAAQAtbqDsWVZQ_L1CZHU621q0Js5LIqjj3zt9zUndLnKo5fwAodAAAAAAwAA
	*BMXBAAQA1HYQpF6zZvbANVzMItmXgBUXcRSOrZ0oqVUT1keT2HD0AodAAAAAAwAA
SystemUpdate	Microsoft-CryptoAPI/5.131.2600.2180
	Microsoft-CryptoAPI/5.131.2600.5512
P2P	BTWebClient/2000(17920)
	uTorrent/1830(15638)
Unknown	C470IP021910000000
	#2YX!!!!#=A@io!#3RM!!!#U=Q7fV!#3

have tested by varying the value from 1 hour, 4, 8, 12 and 24 hours. The results do not fundamentally change very much — this is because the hostnames associated with the flows tend to be fairly stable. The only difference is that when using smaller values, the size of flow groups is smaller. Our system currently runs as an offline UA string classification system. Clearly, when running the system in an “online” manner, a smaller value is likely preferred for fast classification. However, in such a case, one can retain and utilize historical observations of the UA string-hostname associations to make prediction. Investigating this subject is outside the scope of the current paper.

5.1. Algorithm Description

The hostname association is a two step process. In the first step, we compute the entropy of the non-standard UA string by using a pseudorandom number sequence test program [16] to identify those that are likely to be a SHA or MD5 hash. If the entropy is more than four bits/byte, we retain this string in the “flow grouper” (see Figure 1) for a period of time. In the second step, for those UA strings stored in the flow grouper, we count the number of top-two level domain names (extracted from the HTTP hostname field) associated with each UA string. If the number of top-two level domain names is equal to a certain threshold (e.g. we chose 1 for software update), we consider this UA string as associated with a specific application and assign the label based on the corresponding top-two level domain name.

Prepared using nemauth.cls
DOI: 10.1002/nem

5.2. Evaluation

To evaluate this algorithm, we first build our ground truth. For this, we analyzed our dataset with the CFG-based parsers in order to identify those non-standard UA strings. After this, we manually identified the cases that are AV and then compared them with the results from our algorithm. We achieved promising results, with precision of 0.9039 and recall of 0.9463. Figure 2 shows the Complementary Cumulative Distribution Function (CCDF) and the number of unknown UA strings after (i) applying the BNF parsing only, and (ii) after using the hostname association as well. The results show that we effectively reduce the number of unknown UA strings using our proposed heuristics. The percentage of clients having more than five unknown UA strings reduced from 80% to 50% after using the hostname association.

5.3. Data Analysis

In our dataset, non-standard UA strings $(62,615)$ pass through the hostname association to filter application-aware cases $(58,522, 61.7\%)$, where the majority are AV $(53,448)$. More detailed statistics are shown in Table VI.

Table VI. Hostname association for non-standard UA strings.

Category	Hostname-based Association										
	AV			SystemUpdate			P2P			Other	Total
	Norton	AVG	Other	Window	Google	Other	BT	uTorrent	Other		
UAs	42,552	3,937	6,959	1,143	769	113	120	84	78	2,860	58,522
Flows	55,910	68,233	1,391,339	1,373,779	937,834	182,381	63,921	93,425	132,834	1,373,779	5,673,435

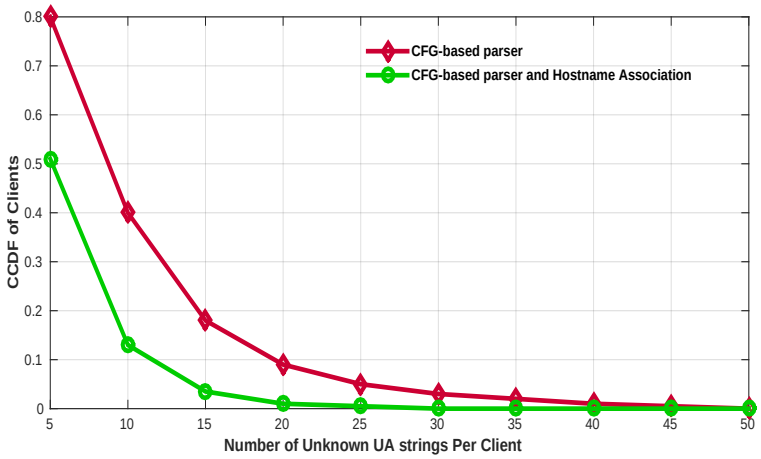

Figure 2. Unknown UA strings after different classification schemes.

Prepared using nemauth.cls
DOI: 10.1002/nem

6. MALICIOUS UA DETECTION

In this section, we demonstrate the utility of the UA analysis in identifying anomalies in host activity and in detecting malicious activities. As a proof of concept, we provide three basic inference engines: fake UA string detection, fake OS detection, anomaly detection, for detecting suspicious standard UA strings, and signature based inference engine for detecting suspicious non-standard UA strings.

We note that parsing process described in above sections by itself does not tell us whether a UA string is fake or not, as a UA string which can pass the standard format check can still be fake. This is where the application profiles extracted from the UA strings come into play. Given HTTP flows generated by a single client IP address that is associated with a single device to separate and group flows based on devices, we build application profiles based on the information extracted from the UA strings, which tell us what OS (and its version) and common applications (and their versions) are running on the device. These application profiles are used by the inference engines to detect fake (or generally malicious) UA strings. In particular, we look for conflicting information contained in the application profiles. Clearly, detecting fake UA strings (especially when they pass the standard format check) hinges on many contextual and other information contained in the HTTP flows, and so forth. For example, in general given a Window machine, only one version/instance of the IE browser can be running. When we see multiple versions of IE are running, or a version of IE is running on the wrong OS version, this signifies that some of these are fake.

To evaluate these inference engines, we first build our ground truth. We primarily use the commercial IDS to help us separate client machines that are infected with malware vs. those that are not. We use the UA strings extracted from HTTP flows generated by standard applications running on the "clean" machines as samples for context-free grammar specifications. We also check the formats obtained from these samples against those generated from test machines as well as other sources of UA lists. Apart from this, we also use the malware labels generated by the IDS to help us confirm the fake/malicious UA strings we have detected by our system. More specific, we apply inference engines to analyzing clients infected by Backdoor.Tidserv (aka Tidserv). Tidserv is a

Prepared using nemauth.cls
DOI: 10.1002/nem

Trojan horse that displays advertisements, redirects user search results, and opens back doors [4]. There are 14 clients which have already been labeled as Tidserv by a commercial IDS in our dataset.

We hope that the readers of our paper can see the utility of our methodology from the inference engines we designed. Based on the methodology described above, security analysts can develop their own heuristics according to the concrete problem they are targeting. We begin by presenting suspicious standard UA strings and then continue to show suspicious non-standard UA strings.

6.1. Suspicious Standard UA Strings

Fake User-Agent Detection. In inference engine, fake UA detection, we aim to detect fake UA strings which contains conflicting information in UA string itself. As we know, a significant number of UA strings can pass the BNF grammar defined in Section 4 and be classified as standard UA strings. However, not all of them are valid as mentioned above. From the left and middle parts of Figure 3, we can see that the number of UA strings and Browser type UA strings in the 14 Tidserv clients are more than those in 100 randomly selected clean clients. This indicates the possibility of fraudulent UA strings generated by Tidserv clients. Given the UA string in Table VII, we suppose it to be generated by a Firefox browser version 2.0. According to the Firefox official site, however, we find that Firefox 2.0 is supported by Windows 98 and other recent OS versions, but not by Windows 95. Nevertheless, the UA string is correctly parsed by our CFG.

Moreover, consider the case of "browser-prefix" and "rendering-engine" rules defined in Table II. Note that in practice, not all browsers are valid with all rendering engines. For example, browser MSIE is only associated with engine "Trident", and if it is associated with another engine such as Gecko or Presto, the UA string tends to be fraudulent.

Combining the aforementioned cases, in order to improve our system, we plan to develope a basic type checking system that checks the dependency between keywords in different terms. This idea is borrowed from runtime type checking in compilers. The linkages can be created between terms by crawling sites, such as [9], to obtain all possible valid combinations of terms and the type checker

Prepared using nemauth.cls
DOI: 10.1002/nem

can enforce them after the BNF parsing. Administrators can also specify their own dependencies in the type checking system.

Table VII. Example suspicious UA strings.

		Examples
Suspicious standard UA strings	fake UA detection	`Mozilla/5.0 (Windows; U ; Win95; it; rv: 1.8.1) Gecko/20061010 Firefox/2.0`
	fake OS detection	`Mozilla/4.0 (compatible; MSIE 6.0b; Windows NT 5.0; .NET CLR 1.0.2914)`
	anomaly detection	`Mozilla/5.0 (Windows NT 5.1) AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.152 Safari/535.19`
Suspicious non-standard UA strings	suspicious standard UA strings	`User-Agent: NULL`
		`Trojan Brontok A11`
		`MSlE`

Fake Operating System Detection. In inference engine, fake OS detection, we aim to detect fake UA strings which contains conflicting information with contextual information contained in the HTTP flows, particularly OS information contained in HTTP flows. OS information can either be inferred from most standard UA strings or OS fingerprinting mechanisms by utilizing pieces of information contained in lower layers (below application layer in ISO model). However, the OS information inferred from UA strings and lower layers could be different. This gives us the indication that UA string could be fake. From the right side of Figure 3, we can see the number of OSes inferred from the UA strings in Tidserv clients is more than that in clean clients. This indicates that the OSes information inferred from fraudulent UA strings might not match the actual OS of the device that generated the corresponding HTTP flow. In order to check whether there are such OS conflicts, we resort to OS fingerprinting mechanisms. The tool we use is "p0f v3", which utilizes an array of sophisticated, purely passive traffic fingerprinting mechanisms to identify the players behind any incidental TCP/IP communications [10].

We ran p0f through Tidserv clients and identified many inconsistent HTTP flows, where the OS inferred from the UA string by our parser is different from the OS provided by p0f. However, such OS conflicts are not found in clean clients. For example, given the UA string in Table VII, it is supposed to be generated by a Windows 2000 machine according to its NT version. However, from the p0f results, the HTTP flow is generated by a Window 7 machine. To further verify this (since p0f might be wrong), we manually went through all the HTTP flows generated from this Tidserv client

Prepared using nemauth.cls
DOI: 10.1002/nem

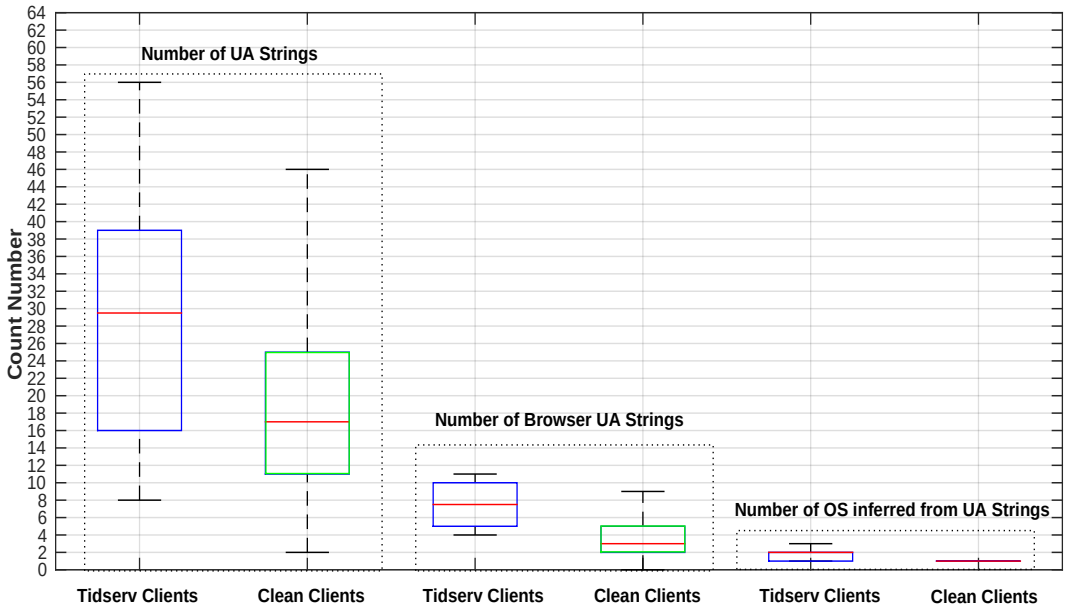

Figure 3. Number of (UA & Browser UA & OS) in Tidserv and clean clients.

and found that no other flows were associated with Windows 2000. In addition, by doing a HTTP referer analysis on this flow, we found that the referred hostname associated with this UA string was never accessed by the monitored client, which indicates a potential click-fraud event. Surprisingly, we found that all 6 clients infected with both Tidserv and Trojan.Zeroaccess [5], generate all the UA strings with OS conflicts. We found a total of 8 distinct fake OSes. We hypothesize that Zeroaccess has a simple codebase that randomly picks a UA from standard UA strings pool without checking the OS of the device that is running the malware.

Anomaly Detection. In inference engine, anomaly detection, we aim to detect fake UA strings from statistic analysis. It is possible for fraudulent UA strings to pass the CFG-based parser, the fake UA detection and the fake OS detection. In this case, we rely on further statistical analysis to identify suspicious hosts. Those fraudulent UA strings can also be found in the Tidserv infected clients. From the statistics of the 14 Tidserv clients in our dataset, the number of standard browsers in those clients is larger than that in clean clients as shown in Figure 3. This indicates a suspicious behavior, as in normal cases, we expect very few browsers being used in a single household.

To better understand the reason for the large number of standard browser UA strings in Tidserv clients, we chose a Tidserv client and digged into it. In this client, there were 12 standard browser

Prepared using nemauth.cls
DOI: 10.1002/nem

UA strings, including different versions of Chrome, RockMelt and Internet Explorer. We found that one of the Chrome UA strings as shown in Table VII was the most frequently used and default for that client, but other UA strings belonged to browsers that were used sporadically and appeared only around flows flagged by the commercial IDS as Tidserv. Investigation showed that the flows with RockMelt and some other standard browser UA strings were directed to advertisement networks (e.g., ad.zanox.com and ad.doubleclick.net) to perform click fraud. This shows that a statistical analysis on top of the presented UA analysis can indeed help identify anomalous clients.

6.2. Suspicious Non-Standard UA Strings

For non-standard UA strings, studies [14] [23] show that some of them are often associated with the malicious activities. Thus, we design signature based inference engine to detect suspicious UA strings in non-standard UA strings. After hostname association, if non-standard UA strings cannot be associated with well-known applications, they are classified into "Unknown UA Strings". For UA strings in this category, our system depends on the signature collected from domain knowledge and other sources of information, to judge whether they are normal or not. In our system, we set basic signatures collected from various online sources like [17].

As shown in Table VII, these UA strings are filterd by signature based inference engine in Tidserv clients. Since and HTTP server would not be able to perform any improvements on user experience based that UA string, "User-Agent: NULL" is abnormal. For the HTTP flows associated with this UA string, VirusTotal [3] reported that the associated hostname is a malicious software download site. Another example in Tidserv clients was the UA string "Trojan Brontok A11", which infects Windows machines. Note that the bot name is written in the UA string, which could be used as a signal for the C&C server to identify flows from infected clients. The third example in Tidserv clients was a mis-spelling in the UA string. The UA contains "MSlE" (note the lower case "L") instead of "MSIE" (the upper case "I"). After searching for this UA string, we found it associated with malware Troj/Agent-VUD.

Prepared using nemauth.cls
DOI: 10.1002/nem

We can envisage that our system can be augmented with rich signatures in the UA strings that do not pass our CFG-based and hostname association UA string classifiers, such as XSS, SQL injection attacks that are embedded in the UA strings. Such a system may prevent a client machine to launch XSS, SQL injections to a targeted server. If running on the server side, our system can simply only allows HTTP requests with legitimate standard web browsers or mobile apps to be forwarded to the server under protection, thereby filtering out ill-formed UA strings (which contain, e.g., malicious SQL commands for compromising a vulnerable server).

7. CONCLUSION

Most cyber attacks today are performed over HTTP and the UA string carries a lot of critical information that can be leveraged to detect them. We presented a system that identifies fraudulent UA strings by categorizing the strings based on their syntactic format and running a set of inference engines. We classified the standard UA strings with a novel *grammar-guided* approach, which leverages context-free grammar to parse and extract application name, device and operating system information. In addition, we developed a heuristic to classify non-standard UA strings by associating them with the hostname field of the corresponding HTTP flow. We devised three inference engines to identify fraudulent UA strings: fake UA string detection, fake OS detection and anomaly detection using statistical features. Finally, we provided several case studies to demonstrate how our approach can lead us to identify malicious applications.

We recognize that our proposed UA string classification methodology is only meant to be one useful tool of a larger arsenal of tools that a security analyst can leverage for detecting malicious activities and attacks. This is because by only considering the UA string of HTTP flows for anomaly detection, we may miss HTTP attacks where the UA string is not the key indicator. In addition, for the case of non-standard UA strings, we currently provide a very basic level of inference, which is not enough to differentiate benign from malicious cases and requires human analysis. Nevertheless, in this paper, we have described and showed the potential of a methodology capable of highlighting

Prepared using nemauth.cls
DOI: 10.1002/nem

abnormalities in the HTTP behavior of clients by focusing on UA string analysis, even when the anomalies are very subtle (e.g. valid UA strings but conflicts in the OS advertised).

REFERENCES

1. Bison, a YACC-compatible parser generator. http://dinosaur.compilertools.net/bison/.
2. Flex, a fast scanner generator. http://dinosaur.compilertools.net/flex/.
3. VirusTotal. https://www.virustotal.com/.
4. Backdoor.tidserv. http://www.symantec.com/security_response/writeup.jsp?docid=2008-091809-0911-99, Nov 2013.
5. Trojan.zeroaccess. http://www.symantec.com/security_response/writeup.jsp?docid=2011-071314-0410-99, Nov 2013.
6. Browser capabilities project. http://browscap.org/, Sep 2014.
7. Browserscope project. https://github.com/tobie/ua-parser, Oct 2014.
8. Hypertext transfer protocol (http/1.1): Semantics and content. https://tools.ietf.org/html/rfc7231, June 2014.
9. List of user agent strings. http://www.useragentstring.com/pages/useragentstring.php, 2014.
10. p0f v3. http://lcamtuf.coredump.cx/p0f3/, Dec 2014.
11. Shellshock. http://en.wikipedia.org/wiki/Shellshock_(software_bug), Sep 2014.
12. User agents in botnets. http://www.behindthefirewalls.com/2013/11/the-importance-of-user-agent-in-botnets.html, Oct 2014.
13. Useragentstring website. http://www.useragentstring.com/, Oct 2014.
14. Advanced sql injection on user agent. http://sechow.com/bricks/docs/content-page-4.html, Mar 2015.
15. *Piwik detects Opera 15 as Chrome 28*. May 2015.
16. A pseudorandom number sequence test program. http://www.fourmilab.ch/random/, Mar 2015.
17. *Reverse Engineering a D-Link Backdoor*. May 2015.
18. M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th International Conference on World Wide Web*, 2010.
19. I. Fette, N. Sadeh, and A. Tomasic. Learning to detect phishing emails. In *Proceedings of the 16th International Conference on World Wide Web*, 2007.
20. G. Gu, J. Zhang, and W. Lee. BotSniffer: Detecting botnet command and control channels in network traffic. In *Proc. of the Network and Distributed System Security Symposium*, 2008.

Prepared using nemauth.cls
DOI: 10.1002/nem

21. R. Holley and D. Rosenfeld. Maul: Machine agent user learning. `http://cs229.stanford.edu/proj2010/HolleyRosenfeld-MAUL.pdf`, Dec 2010.

22. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, chapter 4: Context-Free Grammars, pages 77–106. Addision Wesley, 1979.

23. N. Kheir. Analyzing http user agent anomalies for malware detection. In *Data Privacy Management and Autonomous Spontaneous Security*. Springer, 2013.

24. N. Kshetri. The economics of click fraud. *IEEE Security & Privacy*, 8(3):45–53, May 2010.

25. Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman. Identifying diverse usage behaviors of smartphone apps. In *Proceedings of ACM Conference Internet Measurement Conference*, 2011.

Prepared using nemauth.cls
DOI: 10.1002/nem