INTERNATIONAL JOURNAL OF NETWORK MANAGEMENT

Int. J. Network Mgmt 2015; **00**:1–22

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[2], Alok Tongaonkar[2] and Marco Mellia[3]

[1]*University of Minnesota*

[2]*Narus, Inc.*

[3]*Politecnico di Torino*

SUMMARY

Security is one of the important aspects of network management. 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.

Copyright © 2015 John Wiley & Sons, Ltd.

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

Copyright © 2015 John Wiley & Sons, Ltd.

1. INTRODUCTION

HTTP has become the *defacto* application-layer "transport" protocol, over which many applications such as JSON, SOAP, RSS, 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 [15], phishing [16], botnet command-and-control (C&C) [17], click frauds [22], and so forth.

Given a collection of HTTP flow traces (i.e., network traffic over TCP port 80) passively captured within a network (e.g., at a gateway router) where HTTP and TCP/IP header information are collected (but not application payload), 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 and suspicious activities. 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 browser type, version, operating system (OS), device model, and other related information. Web servers utilize such information to customize their response to the web browser for proper rendering. Due to the diversity of browser types and versions, the state-of-the-art mechanisms for processing UA strings utilize a set of (*ad hoc*) pattern matching heuristics based on a collection of regular expressions (regexes) (see, e.g., [8]). These mechanisms are designed for web servers to recognize web browsers for rendering purposes, *not* for separating normal web browsers from anomalous ones. They are generally ineffective in recognizing *non* web-browser applications that often include partial information (e.g., only the name of an application but

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

no OS or device type) in an application-specific format or even a random-looking character string (see Section 5 for examples). The UA field has also been utilized for malware signature generation for malware detection [20, 21]. These approaches suffer from some of the common problems of any signature-based malware detection systems: (i) a large set of signatures need to be maintained corresponding to different types of malware, and (ii) attackers can bypass these systems by using polymorphism, i.e., using modified UA strings that are not part of the signature set.

We present a novel way of using UA fields to robustly separate and classify applications transported over HTTP, with the goal of detecting malicious applications and activities. Similar to anomaly based system, we build profiles of various well-known applications on different platforms in terms of UA strings. We represent these profiles as context-free grammars (CFGs) [19]. This allows us to use standard compiler technology for efficiently and robustly parsing the UA strings to identify normal applications such as standard web browsers, iOS and Android apps. 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 augment our information extraction engine to infer information for a UA string with hostname association rules. Hostname association identifies UA strings that lack structure, and are always associated with the same domain name (e.g. same antivirus website `kaspersky.com`).

Our contributions are three-fold:

1. We develop a baseline system for generating grammar based profiles of UA strings for "standard" applications over HTTP.
2. We extend this baseline system into a full UA-classification and anomaly detection system by incorporating novel techniques for extracting information from UA strings – using grammar profiles and hostname association rules – that are used to detect a variety of malicious activities in HTTP traffic.
3. We apply these mechanisms to a 24-hour network trace from a nation-wide Internet Service Provider (ISP), and demonstrate that our system is capable of detecting various attacks with

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

different artifacts such as non-standard UA strings, and fraudulent UAs that were used to mimic standard UAs.

Roadmap. The remainder of this paper is organized as follows: We give the background of UA strings, related work in parsing UA strings and the description of our dataset in section 2. The workflow of the system is illustrated in section 3. Given a bunch of UA strings, some of them are standard while the others are non-standard. For standard UA strings, application profile is used to filter them and extract OS, device and application type from those well-structured UA strings, the detail of which is illustrated in section 4. For non-standard UA strings, hostname association is applied, the detail of which is illustrated in section 5. After that, in section 6, we demonstrate the utility of extracted information like OS, device and application type in identifying anomalies in host activity and in detecting malicious activities. Finally, we conclude our paper and have a discussion on the limitation of our method.

2. BACKGROUND, RELATED WORK AND DATASET

2.1. Background

The User-Agent (UA) field of the HTTP header is a string that is often used by a client to communicate its operating parameters to an HTTP server. These parameters could include the client's operating system, browser type and version, the rendering engine, and the application name in the case of traffic from mobile devices [23]. However, the UA string is also used by malware for illicit activities, for instance, as a way to spoof a legit 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 [13]. More recently, the UA string has been used as a way to exploit servers vulnerable to the Shellshock [12] attack.

The big challenge for the network administrator and security analyst is that it is difficult to differentiate between legit and malicious UA strings. The reasons are two-fold. First, although there is a standard UA format defined in RFC 7231 [9], not all benign applications follow it (e.g. Table V),

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

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.

We present a system that identifies malicious UA strings. We first create UA application profiles for various applications that use standard UA formats, such as different browser types as well as iOS and Android apps. These application profiles are escentially parsers that are built using off-the-shelf compiler techniques. In addition, for applications that use non-standard UA string formats, such as Antivirus software, we associate UA strings to benign applications based on the hostname in the header of the HTTP connection. Finally, we use this information and develop analytics to identify malicious cases.

2.2. 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 to identify the client browsers to supply the appropriate web content to them. 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 [7]). Others combine such methods with *regular expression* based classification rules [8, 14], where a multi-language parser based on a laundry list of regular expression (supplied by different people and accumulated over time) is developed. Our experience in using this tool reveals that these tools often produce incorrect classification results. Due to the list of complex regexes it relies on, it is very difficult to debug and manage.

This motivated us to use MAUL [18], which is a machine learning (ML) based UA classification scheme. The problem with ML based schemes is the 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, our context-free grammar based classification scheme not only classifies standard UA

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

strings more accurately, but also detects syntactic errors and anomalies in (invalid) browser-like UA strings.

Finally, UA strings have also been utilized to detect malicious activities, e.g., for detecting SQL injection attack [4]. In particular, Kheir [21] 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.

2.3. Dataset

We use a 24-hour dataset from a large, nation-wide ISP in April 2012. 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.

3. SYSTEM OVERVIEW

Our system has three main steps; (i) application profile generation; (ii) determination of standard and non-standard UA strings and (iii) malicious UA strings inference. In the first step, we develop a UA application profile, as presented in Figure 1. We take a set of training UA strings and pass them through a regex-based UA parser module, such as [8]. For those UA strings for which the application can be extracted, we group them by the application name. Next, we manually clean the UA strings grouping from false positives, since the regex-based parser often creates false positives, such as confusing HTTP flows from mobile applications with those of popular browsers. Next, we manually generate the UA application profile, which is described in the next section. Briefly, the UA application profile is a manually generated grammar to parse the UA strings from those applications,

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

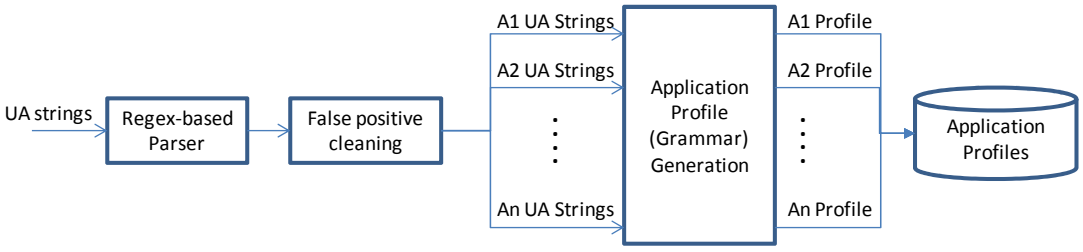

Figure 1. Application profile (grammar) generation.

such as popular browsers (Internet Explorer, Chrome and Firefox) and iOS and Android apps. All application profiles are stored to be used in the next steps.

In the second step, we pass the UA strings through our application profiles as shown in Figure 2. If a UA string matches any of the profiles, we consider it a standard UA and extract the application name, OS and device (if present) from it. Otherwise, the UA string is non-standard.

In the final step, standard UA strings are passed through the inference engines. As described in Section 6, there are three engines. The first one identifies fake UA strings. It checks that the versions of OS and browser are valid and compatible with each other. The second engine identifies cases where the OS extracted from the UA string and the OS inferred from Layer-3 and Layer-4 from the OSI model are in agreement. The third inference engine performs statistical analysis for anomaly detection to identify malicious behaviors in groups of UA strings coming from a client. In the case of non-standard UA strings, they are stored in the flow grouper together with the hostname for that particular HTTP flow. Flows are collected for a period of time until they can be associated to an application name inferred from the hostname, as described in Section 5. If a UA string is associated to a benign standard UA string, it is considered benign. All other UA strings are considered suspicious and further analyzed.

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 brower-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

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

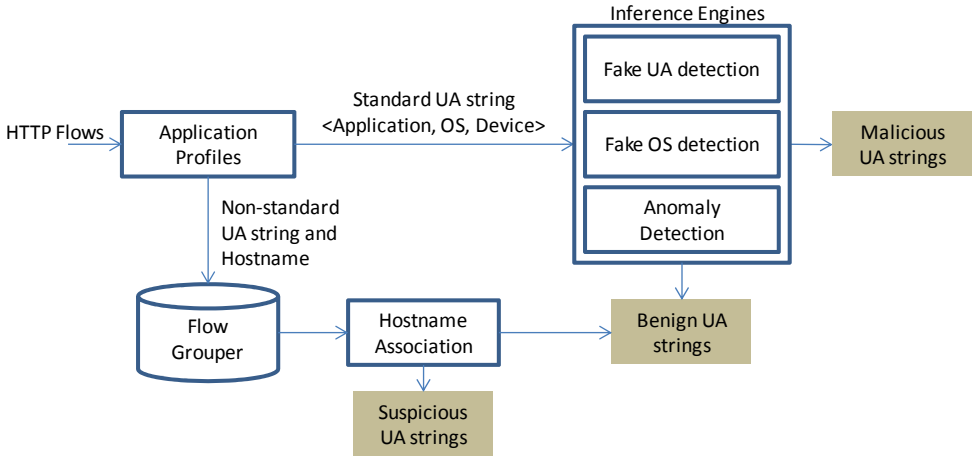

Figure 2. System architecture.

parser, which is a series of per-application context free grammars, which we also call application profiles throught the paper.

4.1. Parsing UA Strings with Regular Expressions

A straightforward solution to this problem is to build a huge list of regular expressions (regexes) that encodes all possible UA strings in the Internet today. In such a system, an incoming UA string is passed through the list of regexes and the first one that matches is used to extract any key information fromt the string. BrowserScope [8] is an example of this approach. However, this method has several problems:

An innapropiate 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 performace. In most cases, a single standard UA string will be tested against multiple regexes before finding a match, since there is no optimization in the regexes to test. Furthermore, in the case of non-standard UA strings, all the

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/nem

regexes in the long list might have to be tested. 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 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 parsed by regular expressions. In our datasets, we found the case of embedded web browsers 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 [19].

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

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 that 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 generated by these 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.

Copyright © 2015 John Wiley & Sons, Ltd.

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

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 datasets. 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 “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

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

Table II. Example production rules for browser UA strings.

```
<standard-browser>::=<browser-prefix>"("<OS-sytem>")"<browser-suffix>;
<browser-prefix>::="Mozilla/4.0"|"Mozilla/5.0"|"Opera/"<version-1-dot>;
<browser-suffix>::=""|<render-engine><browser-type>|<browser-type>|...;
<render-engine>::="Gecko/"<ver-no-dot>|"AppleWebKit/"<ver-1-dot>|...;
<OS-system>::="compatible;"<IE><window><IE-suffix>|<window>|<OSX>|...|...;
<window>::="Window NT "<version>[ <additional-window-info>];
...;
<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";
```

same keywords (most commonly with “Mozilla/5.0”), followed by a set of OS-related keywords enclosed by “(...)”, and at the end, 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 above examples, 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 with 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

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

$<\ldots>$ 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 " " denote a white space, and "|..." indicate that there are additional rules that are not specified due to the space limitation.

Leveraging existing compiler tools *Flex* [2] and *Bison* [1], we develop a "compiler" to parse the UA strings generated by these standard web browsers, classify and extract the browser type, OS, device and other relevant information. The compiler 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 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 compiler 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., [8]), the resulting production rules are easy 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. In other words, just because a UA string satisfies the grammar rules does not necessarily imply that it is a *valid* UA string generated by a standard browser. For instance, the prefix "Mozilla/4.0" is only associated with certain (older) IE versions; and for a specific language version of a browser, only certain version numbers might be valid. Such semantic constraints can be verified at the last step by invoking appropriate browser verification modules based on the browser type, OS and other relevant information extracted by the

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

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.

UA Strings from iOS/MacOS Apps and Other "Well-Known" 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.

Table III. 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				

Evaluation. To evaluate the effectiveness and efficiency of our BNF-based algorithm, we compare our approach with the state-of-art, regexes, in UA string processing. UA strings are randomly chosen from our dataset and passed through both approaches. Then the running time of each approach is recorded, shown in Table IV. For parsing the same amount of UA strings, BNF-based approaches are much more efficient than regexes. The reason why the performance of regexes is not good is because a linear scan over many regexes is required and degrades performace. Detailed explanation is in Section 4.1.

Table IV. Parsing time (in second) of UA strings using CFG vs. regexes

	1	2	5	10	50	100	1,000	10,000	100,000
CFG	0.001	0.001	0.001	0.001	0.001	0.002	0.005	0.042	0.393
Regexes	0.323	0.324	0.328	0.332	0.343	0.359	0.709	4.097	38.211

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 separate these

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

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

AntiVirus	*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

unique UA strings into two categories: *standard* UA strings ($32,261$ unique UA strings, about 34%), which match one of the BNF parsers (i.e., follow well-defined syntaxes), and *non-standard* UA strings ($62,615$ unique strings, about 66%) that do not match any of the BNF parsers. Of the standard UA strings, $23,298$ (24.6%) of them match parsers for browsers and related applications (e.g. browser plug-ins, Window Live, Window Media Center, etc) and $8,963$ (9.4%) match *non-browser* parsers for iOS/MacOS, Android and other well-known non-browser applications with well-defined syntaxes. Note that not all of these standard UA strings are necessarily *valid*, and are in fact “fake” ones generated by malicious applications that mimic the legitimate browser and other applications (see Section 6). In Table III we provide detailed statistics of our classification results. In this table, the “other” category in browser-type UA strings includes browsers such as Sea Monkey, Rockmelt, and other less popular browsers and related browser plug-ins and applications. Similarly, the “other” category for non browser UA strings includes traffic from mobile devices such as Blackberry, and gaming consoles such as Xbox and Playstation.

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, antivirus signature updates and OS updates lack structure and contain random strings. To assign labels to user-agents in this category, we develop a heuristic that we call hostname-based association.

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

We analyzed our datasets and noticed that many of the non-standard UA strings belong to specific applications, such as antivirus (AV) software. AV software embeds local information (e.g. software version, signatures database version, etc) in the UA HTTP header when checking for signature updates. Those UAs differ from browser UAs in two aspects: (i) each UA 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 header field is one or two unique top-two level domain name e.g., all flows in this category communicate with a specific antivirus such as `kaspersky.com` or `kaspersky.net`. Operating system updates (e.g. for Windows and Mac), exhibit similar properties, but they are less random as they do not include any hash of the data. Our heuristic, which we describe below, tries to cover both AV and OS updates cases.

Algorithm Description. The hostname association is a two step process. In the first step, we compute the entropy of the non-standard UA string 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 2) 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 one (1), we consider this UA string in the software updates category and assign the label based on the corresponding top-two level domain name.

Evaluation. To evaluate this algorithm, we first build our groundthruth. For this, we passed our dataset through the ua-parser tool [8] in order to identify those non-stadard UA strings, i.e. cases that don't match any of the regexes in ua-parser. In addition to this, we performed extended manual analysis to clean up the results from the ua-parser tool, since it creates false non-standard cases, such as many UA strings from android devices that fail to match any of their regexes. After this, we manually identify the cases that are software updates and then compare them with the results from our algorithm. We achieved promising results, with precision of 0.9039 and recall of 0.9463.

Data Analysis. Figure 3 shows the 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

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

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

Category	Hostname-based Association									Other	Total
	AntiVirus			SystemUpdate			P2P				
	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

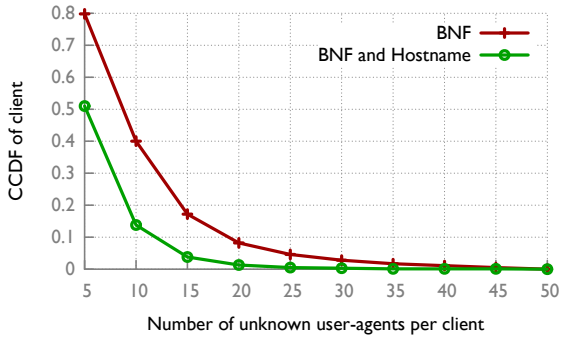

Figure 3. Unknown User-Agents after different classification schemes.

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. A detailed summary is shown in Table VI. In summary, non-standard UA strings $(62,615)$ pass through the hostname association to filter application-aware cases $(58,522,61.7\%)$, where the majority are software updates $(57,097)$. The remaining non-standard UA strings $(4,093)$ are unknown and considered suspicious.

6. MALICIOUS UA DETECTION

In Section 4 and Section 5, we provided a framework to parse UA strings and attach information of device, OS and application type to UA strings. We now demonstrate the utility of these information in identifying anomalies in host activity and in detecting malicious activities. To this end, we describe a series of representative examples found in our dataset. We begin by presenting use cases for suspicious standard UA strings and then we continue to show suspicious non-standard cases. A framework of parsing UA strings and generating application profiles has been proposed in Section 4 and Section 5. We now demonstrate the utility of these information in identifying anomalies in host activity and in detecting malicious activities. As a proof-of-concept, our system is used to analyze clients infected by Backdoor.Tidserv (aka Tidserv). Tidserv is a Trojan horse that uses an advanced

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

rootkit to hide itself. It also displays advertisements, redirects user search results, and opens a back door on the compromised computer [5]. Several flows in those 14 clients have already been labelled as Tidserv by a commerical IDS. 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. A significant number of UA strings can pass the BNF grammar defined in Section 4 and be classifed as standard UA strings. However, not all of them are valid. 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 tends to be fraudulent.

As another example, consider the UA string $Mozilla/5.0(Windows; U; Win95; it; rv : 1.8.1)Gecko/20061010Firefox/2.0$ found in tidserv case. This flow is supposed to be generated by a Firefox browser in 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 CFG.

In order to improve our system, we plan to devise a basic type checking system that checks the dependency between keywords in different terms. This is an idea borrowed from runtime type checking in compilers. Administrators can create linkages between terms by crawling sites such as [10] to obtain all possible valid combinations of terms and the type checker can enforce them after the BNF parsing.

Fake Operating System Detection. OS information can be extracted from most standard UA strings. For example, $Mozilla/4.0(compatible; MSIE6.0; WindowsNT5.1)$ is generated by a Windows XP machine. However, the OS information extracted from fraudulent UA strings might not match the actual OS of the device that generated the corresponding HTTP flow. Thus, we turn to OS fingerprinting mechanisms to check whether there are such OS conflicts. The tool we use is

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

"p0f v3", which utilizes an array of sophisticated, purely passive traffic fingerprinting mechanisms to identify the players behind any incidental TCP/IP communications [11].

We ran p0f through Tidserv clients and identified many inconsistent HTTP flows, where the OS extracted from the UA string by our parser is different from the OS provided by p0f. However, such OS conflicts are not found in the randonly selected clean clients. For example, we found a Tidserv client IP address that generated an HTTP flow with the following UA string: $Mozilla/4.0(compatible; MSIE 6.0b; Windows NT 5.0; .NETCLR 1.0.2914)$. This UA string is supposed to be generated by a Windows 2000 machine according to 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 and found that no other flows were associated with Windows 2000. In addition, by doing a 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 such OS conflicts appear in all six clients infected with both Tidserv and Trojan.Zeroaccess [6] (aka Zeroaccess) in our dataset. 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.

It is possible for fraudulent UA strings to pass the BNF-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 Tidserv threat. From the statistics of 14 clients infected by Tidserv in our dataset, the number of standard browser in those clients (mean: 7.8; median: 6) is larger than that in unflagged (benign) clients (mean: 3.0; median: 2). This indicates a suspicious behavior, as in normal cases, we expect very few browsers being used in a single household.

To better understand reason why there are many standard browser UA strings used in Tidserv clients, we chose a Tidserv client and digged into it. In this client, there are 12 standard browser

Copyright © 2015 John Wiley & Sons, Ltd.

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 is the most frequently used and default for that client, but other UA strings belong to browsers that were used sporadically and appear only around flows flagged by the commerical 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. Such a statistical analysis on top of the result of UA analysis can indeed help security analysts identify anomaly in client host.

6.2. Suspicious Non-Standard UA Strings

We investigated "suspicious" non-standard UA strings that were not associated with benign hostnames. An example UA string is "User-Agent: NULL". This is very abnormal, since the server of the application will not be able to use it to improve user experience. For this HTTP connection, VirusTotal [3] reports the associated hostname is a software download site which is malicious. Another example is the UA string "Trojan Brontok A11", which infects Windows machines. This trojan modifies Windows Registry and Windows Explorer settings and turns off the Windows firewall. 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 is a mis-spelling in the UA string. The UA contains "MSlE" (note the lower case "L") instead of "MSIE" (the upper case "I"). Our research shows that this UA is associated with malware Troj/Agent-VUD. These examples show that our system helps security analysts narrow down the search space of malicious activities.

After hostname association, if non-standard UA strings cannot be associated with well-known applications, they will be classified into "Unknown UA Strings". For UA strings in this category, our system depends on domain knowledge to judge whether it is normal or not. For example, a UA string is found, "User-Agent: NULL", in Tidserv clients. This is very abnormal, since the server of the application is not able to use this UA string to improve user experience. For the HTTP flows associated with this UA string, VirusTotal [3] reports the associated hostname is a software download site which is malicious. Another example in Tidserv clients is the UA string

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

"Trojan Brontok A11", which infects Windows machines. This trojan modifies Windows Registry and Windows Explorer settings and turns off the Windows firewall. 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 is 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. So far, our system has not had a well-defined approach to identify anomaly in unknown UA string, but security analysts can turn to thrir domain knowledge assited with the UA analysis done by our system.

7. CONCLUSION

Most cyber attacks today are performed over HTTP and the User-Agent (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 deviced 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 acknowledge that from UA string itself, we cannot tell much on whether its associated flow is malicious or not. However, whether a UA string is invalid or the information extracted from UA string can provide security analysts with the insights that some hosts are infected by malware. Inference engines in our system are taking use of the analysis of UA strings to help security analysts identify malicious activities. For those non-standard UA strings which are categorized into suspicious set, there is not an effective way of analyzing them in existing system. Those UA

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

string could be malformed due to a buggy implementation of the HTTP stack or given by unknown software vendors. Thus, more sophisticated inference engines for analyzing both standard and non-standard UA strings can be designed and merged into our framework, or expert intelligence can be involved directly with the information provided by our system.

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. Checking for user-agent header sql injection. http://holisticinfosec.blogspot.com/2010/10/checking-for-user-agent-header-sql.html, Oct 2010.
5. Backdoor.tidserv. http://www.symantec.com/security_response/writeup.jsp?docid=2008-091809-0911-99, Nov 2013.
6. Trojan.zeroaccess. http://www.symantec.com/security_response/writeup.jsp?docid=2011-071314-0410-99, Nov 2013.
7. Browser capabilities project. http://browscap.org/, Sep 2014.
8. Browserscope project. https://github.com/tobie/ua-parser, Oct 2014.
9. Hypertext transfer protocol (http/1.1): Semantics and content. https://tools.ietf.org/html/rfc7231, June 2014.
10. List of user agent strings. http://www.useragentstring.com/pages/useragentstring.php, 2014.
11. p0f v3. http://lcamtuf.coredump.cx/p0f3/, Dec 2014.
12. Shellshock. http://en.wikipedia.org/wiki/Shellshock_(software_bug), Sep 2014.
13. User agents in botnets. http://www.behindthefirewalls.com/2013/11/the-importance-of-user-agent-in-botnets.html, Oct 2014.
14. Useragentstring website. http://www.useragentstring.com/, Oct 2014.
15. 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.
16. I. Fette, N. Sadeh, and A. Tomasic. Learning to detect phishing emails. In *Proceedings of the 16th International Conference on World Wide Web*, 2007.
17. 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.

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem

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

19. 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.

20. L. Invernizzi, S.-J. Lee, S. Miskovic, M. Mellia, R. Torres, C. Kruegel, S. Saha, and G. Vigna. Nazca: Detecting Malware Distribution in Large-Scale Networks. In *Proceedings of the ISOC Network and Distributed System Security Symposium*, 2014.

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

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

23. 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.

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nemauth.cls
DOI: 10.1002/nem