INTERNATIONAL JOURNAL OF NETWORK MANAGEMENT
Int. J. Network Mgmt 2015; **25**:306–319
Published online 22 July 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/nem.1900

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, Minneapolis, MN, USA*
[2]*Symantec Corporation, Mountain View, CA, USA*
[3]*Politecnico di Torino, Turin, Italy*
[4]*Korea Advanced Institute of Science and Technology, Daejeon, South Korea*

SUMMARY

Hypertext transfer protocol (HTTP) has become the main protocol to carry out malicious activities. Attackers typically use HTTP for communication with command-and-control servers, click fraud, phishing and other malicious activities, 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, operating system (OS), device, and so on, 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 a proof-of-concept system that demonstrates how our approach can be used to identify malicious applications that generate fake UA strings to engage in fraudulent activities.

Received 19 December 2014; Revised 30 May 2015; Accepted 2 June 2015

1. INTRODUCTION

Hypertext transfer protocol (HTTP) has become the *de facto* application layer 'transport' protocol, over which many applications such as JSON, SOAP, gaming, voice over IP, 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 [1], phishing [2], botnet command-and-control (C&C) [3], click frauds [4] 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 and Chrome) and other commonly used applications (e.g. iOS or Android mobile apps), from 'anomalous' applications.

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

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 [5]. 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 C&C server [6]. More recently, the UA string has been used as a way to exploit servers vulnerable to the Shellshock [7] attack.

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 twofold: (i) although there is a standard UA format defined in RFC 7231 [8], not all benign applications follow it (e.g. Table 5), which limits the possibility of filtering HTTP connections with non-standard UA strings; and (ii) if a malicious UA string is following the standard format, there is no obvious way to detect it.

Because of 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 (e.g. [9]). These tools are designed for web servers to recognize browsers for content rendering, *not* for separating normal browsing behaviour from anomalous cases. In addition, they are generally ineffective in recognizing *non*-browser applications (refer to 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. Our contributions are threefold: (i) We develop a novel context-free grammar (*CFG*)-*based parser* for classifying UA strings generated by 'standard' applications over HTTP that is modular and more effective than the state-of-the-art regex-based tools (Section 4). (ii) We develop a novel UA string–host name association method for classifying UA strings generated by many 'non-standard' applications (Section 5). (iii) We incorporate these mechanisms into a *proof-of-concept* system (refer to 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 on a 24-h network trace from a nationwide Internet service provider (ISP) and present inference engines to illustrate how various attacks with different artefacts such as non-standard UA strings and 'fake' UA strings that mimic standard cases can be detected (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 browscap.dll used by Windows web servers [10]). Others combine such methods with *regular expression*-based classification rules [9,11], where a laundry list of rules is supplied by individuals and accumulated over time. Our experience in using 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 [12], which is a machine learning (ML)-based UA classification scheme. The problem with such an ML-based scheme 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 legit 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, for example, for detecting structured query language injection attack [13]. In particular, Kheir [14] 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.

DOI: 10.1002/nem

3. OVERVIEW

In this section, we provide a motivation for our work, summarize our solution and present a proof-of-concept system design.

3.1. Motivation

Initially, we noticed that existing methods to classify UA strings are used mainly by web servers to improve page rendering, and they are solely based on regular expressions (regexes). The typical order of operations of these systems is as follows. The regexes are organized in a hierarchy, and there is a parsing library that iterates over them. For a given UA string, in the first level of the hierarchy, the parsing library tries to match a regex for the browser name (e.g. Firefox or Chrome). Then, the library iterates 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. 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 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. Approach

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 two phases: (1) extract UA strings from three sources: sandbox environments, network traces from 'clean' clients (those clients without any alert from the intrusion detection system (IDS) in our dataset and from web sites that list valid UA strings, such as that from the Useragentstring web site [11]; (2) write the CFG using standard UA strings from phase 1. In phase 1, a sandbox environment can be used to run different versions of popular web browsers and their popular plug-ins. Then, the generated UA strings are added to a repository with corresponding metadata, which describes the UA strings in the extracted group (e.g. UA strings for Firefox running on Windows with a Firebug plug-in are added to the repository with metadata 'Firefox, Windows and Firebug'). We also put the UA strings from network traces and web sites into the repository. We consider all UA strings in the repository as standard. In phase 2, we use the standard UA strings in the repository to incrementally write the CFG (e.g. we write CFG for Firefox UA strings with a subset of the UA strings, and we add another group of UA strings until we acquire the complete CFG). Later on, 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 analysing 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 instance, the majority of benign non-standard UA strings in our network traces belong to antivirus (AV) and software updates. These UA strings are highly random and show up only once because it includes a cryptographic hash of information that AV is sending to the remote server. However, AV-related UA strings are always associated with an 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 design

Figure 1 illustrates the design of a proof of concept for our system. The input to the CFG engine is the set of HTTP flows per client IP, and the output is the set of standard and 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 the top two level domain names,

DOI: 10.1002/nem

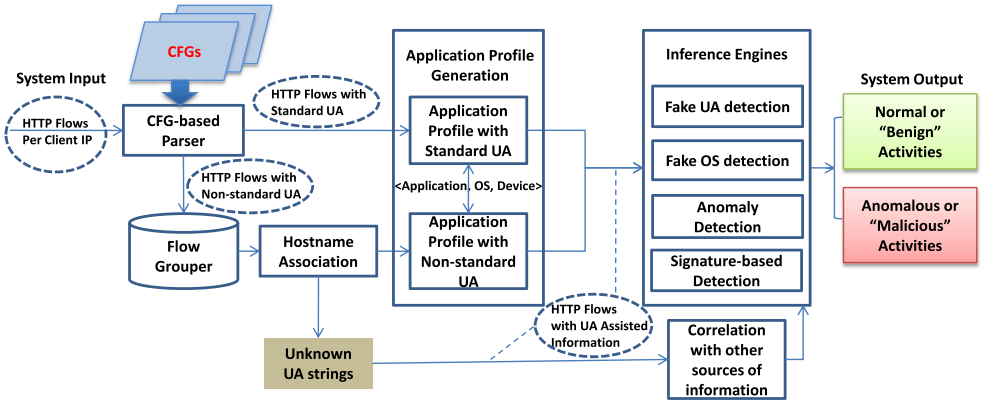

Figure 1. System architecture

for example, all UA strings with domain names `symantec.com` are grouped together. Then, *hostname association* uses hostname information to label UA strings that possess similar characteristics (Section 5). Some UA strings might still be unknown, and therefore, we fall back to other sources of information, such as searching for them on Google. Then, the inference engines are used to search for conflicting or anomalous UA strings (Section 6). Finally, the system produces a report of benign and malicious activities.

3.4. Dataset

We use a 24-h dataset from a large, nationwide ISP collected in April 2012. The monitored network is mostly residential, with high-speed asymmetric digital subscriber line connections to the Internet. The collected data include all inbound/outbound TCP connections to the network. The dataset contains only the TCP and HTTP header information (the HTTP payload was not analysed, 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 labelled by a commercial IDS.

4. STANDARD USER-AGENT 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 CFGs.

4.1. Parsing user-agent 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. 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 [9] 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 mislabelled 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, because

DOI: 10.1002/nem

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, for example, 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 linearly scanning over each individual list is still needed to find different components. In addition, a potential problem of this approach is that the dependencies between device, OS and application are ignored. This causes some fraudulent UA strings to be considered as valid.

Some user-agent 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, such as the following: `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 [15].

To alleviate all these limitations, we build per-application CFG parsers for UA strings, which we describe in the next subsection.

4.2. Parsing user-agent 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 CFGs 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 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 a large amount of UA strings we see in our dataset. Table 1 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,

Table 1. 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/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 (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 (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 (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`
	`Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.0) Opera 12.14`

DOI: 10.1002/nem

Windows OS-related information and the rendering engine 'Trident/[version]' (if present). UA strings of Firefox, Chrome and Safari are structurally similar to IE. On the other hand, the UA strings for Opera browsers begin with 'Opera/[version]', although newer versions also begin with 'Mozilla/5.0'.

In the previously mentioned examples, all standard browser UA strings manifest (nested) matching structures that are characteristic of context-free languages, for example, 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, for example, 'Gecko/[version]' matches only 'Firefox/[version]' or 'Opera/[version]' and 'AppleWebKit/[version]' matches only with 'Safari/[version]'. These (nested) matching structures can be best recognized by pushdown automata, that is, 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 2, 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. 'Version/'). In the previously mentioned examples, note that "" denotes an empty string, while " " denotes a white space, and "|..." indicates that there are additional rules that are not specified because of the space limitation. Readers can find more detailed production rules in our technical report [16].

We leverage existing compiler tools (*Flex* [17] and *Bison* [18]) to 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 analyser* and a *syntax analyser*—and operates in two phases: (i) the lexical analyser first tokenizes a UA string and extracts each meaningful element (i.e. the terminal terms) and (ii) the syntax analyser 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 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 UA parsers relying purely on complex regex-based heuristics (e.g. [9]), our 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) similar to 'type checking' and other run-time 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.

Table 2. Example production rules for browser UA strings

```
<standard-browser>::=<browser-prefix>"("<OS-system>")"<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>|...;
<browser-type>::="Firefox/"<ver-1-more-dot>|"Opera/"<ver-1-dot>|<chrome>|...;
<chrome-safari>::=<chrome-browser>" "<safari-type>|<safari-suffix>;
...;
<opera-version>::="Version/"<ver-1-dot>;
...
<version>::=<ver-no-dot>|<ver-1-more-dot>;
<ver-no-dot>::=<digits>;
<ver-1-dot>::=<digits>"."<digits>;
...;
<digits>::=<digit><digits>:
<digit>::="0"|"1"|...|"9";
```


DOI: 10.1002/nem

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 from the UA string that has passed the syntax analyser. 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 also 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 state-of-the-art regex-based UA string parser [9]. 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 3. For parsing the same amount of UA strings, CFG-based parser is much more efficient than regex-based parser. This is because Browserscope [9] requires a linear scan over many regexes, which degrade performance (refer to Section 4.1 for a detailed explanation).

4.4. Dataset analysis

In our dataset, we find more than 40 million HTTP flows and 94 876 *unique* UA strings. Applying BNF-based UA string parsers for standard applications, we separate these unique UA strings into two categories: *standard* UA strings (32 261 unique UA strings, about 34%), which matches 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. As shown in Table 4, of the standard UA strings, 23 298 (24.6%) of them match parsers for browsers and 8963 (9.4%) match *non-browser* parsers for iOS/MacOS, Android and other applications with well-defined syntaxes.

5. NON-STANDARD USER-AGENT STRINGS

For non-standard UA strings, we develop a heuristic that we call hostname-based association. In the analysis of our dataset, we found that 'non-standard' UA strings roughly fall into two categories: (1)

Table 3. User-agent string parsing time (in seconds) for our CFGs and regexes

No. of 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

CFG, context-free grammar.

Table 4. Standard browser and non-browser UA string classifications

	Browser-type							
Category	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

	Non-browser			
Category	iOS app	Android app	Other	Total
UAs	7 425	871	667	8 963
Flows	1 075 071	56 910	67 244	1 199 225

UA, user-agent.

DOI: 10.1002/nem

Table 5. Example of user-agent strings generated by 'non-standard' applications

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

UA strings containing fairly random strings, composed of alphanumeric characters and (2) UA strings containing certain fixed keywords and some loosely defined structures. In addition, a key observation that we have made is that the hostname field in the HTTP header of flows containing these UA strings provides important hints regarding what type of applications may have generated them. Table 5 shows some examples of this type of UA strings.

5.1. Algorithm

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 [19] to identify those that are likely to be an SHA or MD5 hash. If the entropy is more than 4 bits/byte, we retain this string in the 'flow grouper' (Figure 1) for a period of time. In our testing, we have varied the entropy value from 2 bits/byte to 6 bits/byte and found that 4 bits/byte yields the best result. The flow grouper is used in our system to group all flows with the same top-two level hostname so that we can generate the UA strings to hostname mappings. 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 one for antivirus software), we consider this UA string as associated with a specific application and assign the label based on the corresponding top-two level domain name. For example, if a group of random UA strings belong to HTTP flows where the hostname is kaspersky.com or kaspersky.net, then the application is an AV and the label is 'kaspersky'.

5.2. Evaluation

To evaluate this algorithm, we first build our ground truth. For this, we analysed our dataset with the CFG-based parsers in order to filter out 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 a precision of 0.9039 and a recall of 0.9463. Figure 2 shows the complementary cumulative distribution function of the number of unknown UA strings after applying the CFG-based parser only and in combination with the hostname association. 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, after we applied our hostname association heuristic on non-standard UA strings (62 615), we were able to classify 93.4% of them. The majority of the non-standard UA strings (85.4%) turned out to be AV flows. We show more detailed statistics in Table 6.

6. MALICIOUS USER-AGENT DETECTION

In this section, we show the value of the UA analysis in detecting malicious activities. Our intent is to show examples of inference engines that can be built on top of the methodology described

DOI: 10.1002/nem

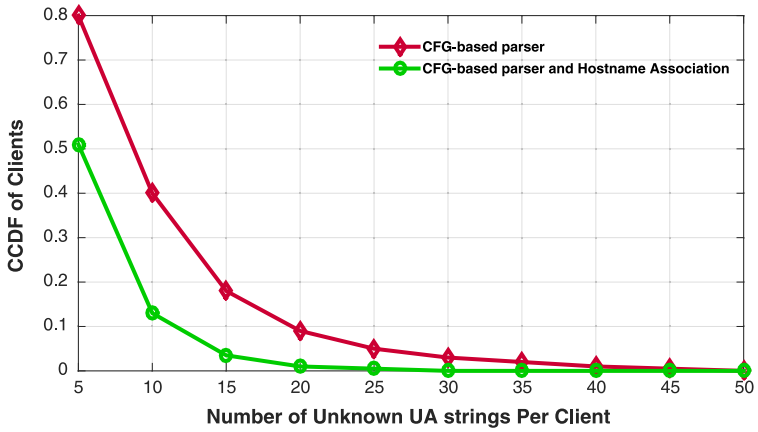

Figure 2. Unknown user-agent (UA) strings after different classification schemes

Table 6. Hostname association for non-standard UA strings

	Hostname-based association										
	AV			System update			P2P				
Category	Norton	AVG	Other	Window	Google	Other	BT	uTorrent	Other	Other	Total
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

previously to demonstrate how security analysts can develop their own inference engines. To detect suspicious standard UA strings, we describe the following engines: fake UA string detection, fake OS detection and anomaly detection. To detect non-standard UA strings, we describe a signature-based detection engine.

We reiterate the fact that parsing by itself does not tell us whether a UA string is fake or not, as a UA string that can pass the standard format check can still be fake. However, inference engines go beyond individual UA strings, by correlating across multiple UA strings generated by individual clients, to detect fake cases. For example, in general, given a Windows client, only one version/instance of the IE browser can be running. However, if we see multiple versions of IE running in a client or a version of IE running on the wrong Windows OS version, these are signs of anomalies in that client.

6.1. Evaluation methodology

To evaluate the inference engines, we first build our ground truth. We primarily use the commercial IDS to help us separate malware-infected client IP addresses (which we will call clients from now on) from 'clean' clients. We apply our inference engines to analyse clients infected by Backdoor.Tidserv (a.k.a. Tidserv) in this evaluation. Tidserv is a Trojan horse that displays advertisements, redirects user search results and opens back doors [20]. There are 14 clients that have already been labelled as Tidserv by a commercial IDS in our dataset.

6.2. Suspicious standard user-agent strings

6.2.1. Fake user-agent detection

Figure 3 shows the box plot of number of UA strings, browser UA strings and OS inferred from UA strings for the 14 Tidserv clients and for 100 clean clients selected at random from our dataset.

From the left and middle parts of the figure, we can see that the number of UA strings and browser UA strings in the 14 Tidserv clients is more than those in 100 clean clients. This is a hint of potential fraudulent UA strings generated by Tidserv clients. For example, consider the UA string in Table 7,

DOI: 10.1002/nem

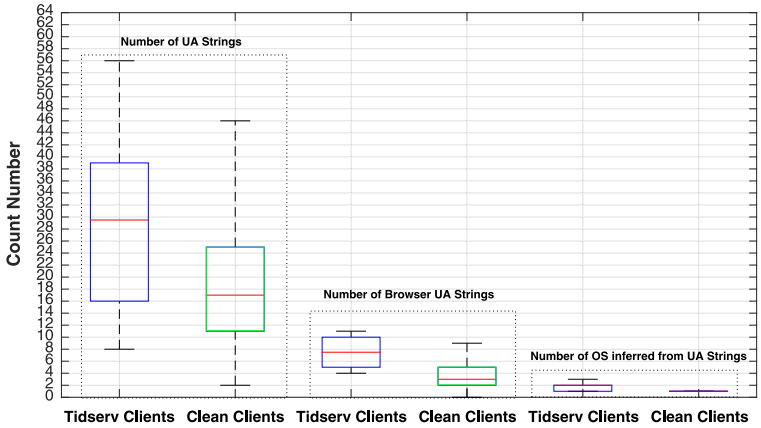

Figure 3. Number of user-agent (UA,) browser UA and operating system (OS) in Tidserv and clean clients

Table 7. Example of 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	Signature-based detection	`User-Agent: NULL`
		`Trojan Brontok A11`
		`MSlE`

UA, user-agent; OS, operating system.

under 'fake UA detection'. This flow is supposed 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 (which is the OS declared in that UA string).

As another example, consider the case of 'browser-type' and 'render-engine' rules defined in Table 2. Note that in practice, not all browsers are compatible 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 has a high chance of being fraudulent.

Leveraging the insights given by the conflicting information described previously, we have developed 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 between terms are created by crawling various sites (such as [21]) to obtain all possible valid combinations of terms, and the type checker can enforce them after the CFG-based parsing. Administrators can also specify their own dependencies in the type checking system.

6.2.2. Fake operating system detection

Fake OS detection aims to find fake UA strings, which contain conflicting information of the OS inferred from the UA string, in comparison with the actual OS running in the client machine. To infer the actual OS running in the client machine, we resort to network-based OS fingerprinting mechanisms. The tool we use is 'p0f v3' [22].

To the rightmost side of Figure 3, we can observe that the number of OSs inferred from the UA strings in Tidserv clients is larger than that in clean clients (even though there are seven times more

DOI: 10.1002/nem

clean clients). This indicates that the OS information inferred from fraudulent UA strings might not match the actual OS of the client machines that generated the corresponding HTTP flow.

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, the UA string in Table 7, under 'fake OS detection', is supposed to be generated by a Windows 2000 machine (indicated by the 'Windows NT 5.0 string'). However, from the p0f results, the HTTP flow is generated by a Windows 7 machine. To further verify this (because 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 analysing the referrer field of the HTTP header for that suspicious flow, we found that the referred uniform resource locator was never accessed by the monitored client, which indicates a potential click fraud event. Surprisingly, we found that all six clients infected with both Tidserv and Trojan.Zeroaccess [23] generate all the UA strings with OS conflicts. We found a total of eight distinct fake OSs. We hypothesize that Zeroaccess has a simple codebase that randomly picks a UA string from a standard UA string pool without checking the OS of the device that is running the malware.

6.2.3. Anomaly detection

Anomaly detection aims to find fake UA strings using statistic analysis. This targets fraudulent UA strings that have passed the CFG-based parser, the fake UA detection and the fake OS detection. 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 hints to a suspicious behaviour, because 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 one of the clients and dug into it. In this client, there were 12 standard browser UA strings, including different versions of Chrome, RockMelt and IE. We found that one of the Chrome UA strings, as shown in Table 7, under 'anomaly detection', 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 string analysis can indeed help identify anomalous clients.

6.3. Suspicious non-standard user-agent strings

6.3.1. Signature-based detection

For non-standard UA strings, some studies [13,14,24] show that malware often embeds malicious information into UA strings. Thus, we design a signature-based engine to find suspicious cases. 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 signatures obtained from domain knowledge and other sources of information to judge whether UA strings are normal or not. In our system, we set basic signatures collected from various online sources, such as from http://www.devttys0.com/2013/10/reverse-engineering-a-d-link-backdoor [24].

Table 7, under 'signature-based detection', shows some examples of malicious UA strings generated by our Tidserv clients and filtered by our engine. For example, for the HTTP flows associated with 'User-Agent: NULL', VirusTotal [25] reports that the associated hostname is a malicious software download site. Another example that we caught was 'Trojan Brontok A11', which is the bot name written in the UA string. It infects Windows machines and can be used as a signal for the C&C server to identify flows from infected clients. Finally, the UA containing 'MSlE' (note the lower case 'L') instead of 'MSIE' (the upper case 'I'), is associated with malware Troj/Agent-VUD and can also be filtered by our engine.

We envisage that our system can be augmented with rich signatures targeting attacks, such as cross-site scripting and structured query language injection embedded in the UA strings. Such a system may

DOI: 10.1002/nem

prevent a client from launching UA-based attacks to a targeted server. If running on the server side, our system can only allow HTTP requests with legitimate UA strings to be forwarded, thereby filtering out suspicious cases.

7. CONCLUSION

Most cyberattacks today are performed over HTTP, and the UA string carries much 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 CFG 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. Finally, we showcase four inference engines intended to demonstrate analytics that can be built on top of our methodology to classify UA strings: fake UA string detection, fake OS detection, anomaly detection using statistical features and signature-based detection.

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 abnormalities in the HTTP behaviour 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). In the future, we plan to research ways of minimizing the human effort required for detecting anomalies from non-standard UA strings.

ACKNOWLEDGEMENTS

This research was supported in part by NSF grants CNS-1117536, CRI-1305237, and CNS-1411636 and DTRA grants HDTRA1-09-0050 and HDTRA1-14-1-0040, and ARO MURI Award W911NF-12-1-0385. We are also grateful to the anonymous reviewers for their valuable comments and suggestions.

REFERENCES

1. Cova M, Kruegel C, Vigna G. Detection and analysis of drive-by-download attacks and malicious javascript code, In *Proceedings of the 19th International Conference on World Wide Web*: Raleigh, North Carolina, USA, 2010; 281–290.
2. Fette I, Sadeh N, Tomasic A. Learning to detect phishing emails, In *Proceedings of the 16th International Conference on World Wide Web*, Banff, Alberta, Canada, 2007; 649–656.
3. Gu G, Zhang J, Lee W. BotSniffer: detecting botnet command and control channels in network traffic, In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA, 2008.
4. Kshetri N. The economics of click fraud, *IEEE Security & Privacy* 2010; **8**(3): 45–53.
5. Xu Q, Erman J, Gerber A, Mao Z, Pang J, Venkataraman S. Identifying diverse usage behaviors of smartphone apps, In *Proceedings of ACM Conference Internet Measurement Conference*, Berlin, Germany, 2011; 329–344.
6. User agents in botnets, 2014. Available from: http://www.behindthefirewalls.com/2013/11/the-importance-of-user-agent-in-botnets.html [9 July 2015].
7. Shellshock, 2014. Available from: http://en.wikipedia.org/wiki/Shellshock_(software_bug) [9 July 2015].
8. Hypertext transfer protocol (http/1.1), 2014. Available from: http://tools.ietf.org/html/rfc7231 [9 July 2015].
9. Browserscope project, 2014. Available from: https://github.com/tobie/ua-parser [9 July 2015].
10. Browser capabilities project, 2014. Available from: http://browscap.org/ [9 July 2015].
11. Useragentstring website, 2014. Available from: http://www.useragentstring.com/ [9 July 2015].
12. Holley R, Rosenfeld D. Maul: machine agent user learning, 2010. Available from: http://cs229.stanford.edu/proj2010/HolleyRosenfeld-MAUL.pdf [9 July 2015].
13. Advanced sql injection on user agent, 2015. Available from: http://sechow.com/bricks/docs/content-page-4.html [9 July 2015].
14. Kheir N. Analyzing http user agent anomalies for malware detection. In *Data Privacy Management and Autonomous Spontaneous Security*, Springer Berlin-Heidelberg: Berlin, Germany, 2013, pp. 187–200.

DOI: 10.1002/nem

15. Hopcroft JE, Ullman JD. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley: Boston, MA, United States, 1979.
16. Technical report: detecting malicious activities with user-agent based profiles, 2015. Available from: http://www-users.cs.umn.edu/~yazhang/materials/technical_report [9 July 2015].
17. Flex, a fast scanner generator, 2015. Available from: http://dinosaur.compilertools.net/flex/ [9 July 2015].
18. Bison, a YACC parser generator, 2015. Available from: http://dinosaur.compilertools.net/bison/ [9 July 2015].
19. A pseudorandom number sequence test program, 2015. Available from: http://www.fourmilab.ch/random/ [9 July 2015].
20. Backdoor.tidserv, 2013. Available from: http://www.symantec.com/security_response/writeup.jsp?docid=2008-091809-0911-99 [9 July 2015].
21. List of user agent strings, 2015. Available from: http://www.useragentstring.com [9 July 2015].
22. p0f v3, 2014. Available from: http://lcamtuf.coredump.cx/p0f3/ [9 July 2015].
23. Trojan.zeroaccess, 2013. Available from: http://www.symantec.com/security_response/writeup.jsp?docid=2011-071314-0410-99 [9 July 2015].
24. Reverse engineering a D-link backdoor, 2015. Available from: http://www.devttys0.com/2013/10/reverse-engineering-a-d-link-backdoor [9 July 2015].
25. Virustotal, 2015. Available from: https://www.virustotal.com/ [9 July 2015].

AUTHORS' BIOGRAPHIES

Yang Zhang received his BS and MS degrees in Software Engineering and Computer Science from Southeast University in 2011 and 2013, respectively. He is currently a PhD student of Computer Science at the University of Minnesota. His research interests lie in network security, software defined network, and innovative networked applications.

Hesham Mekky received his BSc in Computer Science from Alexandria University, Egypt, in 2007 and MSc in Computer Science from the University of Minnesota in 2013. He is currently a PhD candidate of Computer Science at the University of Minnesota. His research interests include networking, security, and privacy. He is a student member of IEEE.

Zhi-Li Zhang received the BS degree in Computer Science from Nanjing University, China, in 1986 and his MS and PhD degrees in Computer Science from the University of Massachusetts in 1992 and 1997. In 1997, he joined the Computer Science and Engineering Faculty at the University of Minnesota, where he is currently a Qwest Chair Professor and Distinguished McKnight University Professor. Dr Zhang's research interests lie broadly in computer communication and networks, Internet technology, and multimedia and emerging applications. He has cochaired several conferences/workshops including IEEE INFOCOM'06 and served on the TPC of numerous conferences/workshops. Dr Zhang is corecipient of five Best Paper Awards and has received a number of other awards. He is a member of ACM and a fellow of IEEE.

Ruben Torres is a principal data scientist at Symantec Corporation. His current research interests are in the areas of network security and big data analytics. In the past, he has also worked in the areas of Internet video distribution, peer-to-peer networks, and traffic classification. He received his PhD (2011) and MS (2006) in Electrical and Computer Engineering from Purdue University and his BS in Electronics and Telecommunications from Universidad de Panama. He has served on the Technical Program Committees of several conferences and workshops, including IEEE Infocom, IEEE ICDCS, and TMA.

Sung-Ju Lee received his PhD in Computer Science from the University of California, Los Angeles (UCLA) in 2000. He spent 12.5 years at Hewlett-Packard Company as a principal research scientist and distinguished mobility architect. He was then a principal member of Technical Staff at the CTO Office of Narus, Inc. In 2015, he joined the faculty of the School of Computing at KAIST. Dr Lee has published over 100 technical papers in peer-reviewed journals and conferences. In addition, he has 32 granted US patents and 40 plus pending patents. He won the HP CEO Innovation Award in 2010 that recognizes the people behind the most innovative products that HP has brought to market. He is an IEEE fellow and an ACM Distinguished Scientist.

Alok Tongaonkar is a data scientist director in the CTO's Office at Symantec Corporation, Mountain View, USA, where he leads research and development activities in the field of big data analytics for network security. He received a Bachelor of Engineering degree from Pune University, India, and MS and PhD degrees in Computer Science from Stony Brook University, NY, USA. In the past, his research focused on applying techniques from compiler domain to problems in network security and management, including aspects such as policy-based systems, performance optimization, and reverse engineering. He has led the development of innovative technology

DOI: 10.1002/nem

in the field of network traffic visibility, including mobile traffic, that has been incorporated in multiple products. He has been a reviewer for several journals and is a senior member of IEEE.

Marco Mellia PhD. His research interests are in the design of energy efficient networks (green networks), in the area of traffic monitoring and analysis, and in cyber monitoring in general. He is the coordinator of the mPlane Integrated Project that focuses on building an intelligent measurement plane for Future Network and Application Management. Marco Mellia has coauthored over 250 papers published in international journals and presented in leading international conferences. He held a position as associate professor at Politecnico di Torino, Italy.

DOI: 10.1002/nem