# ARROW: GenerAting SignatuRes to Detect DRive-By DOWnloads

Junjie Zhang
Georgia Institute of Technology
jjzhang@cc.gatech.edu

Christian Seifert
Microsoft Bing
chriseif@microsoft.com

Jack W. Stokes
Microsoft Research
jstokes@microsoft.com

Wenke Lee
Georgia Institute of Technology
wenke@cc.gatech.edu

## ABSTRACT

A drive-by download attack occurs when a user visits a webpage which attempts to automatically download malware without the user's consent. Attackers sometimes use a *malware distribution network* (MDN) to manage a large number of malicious webpages, exploits, and malware executables. In this paper, we provide a new method to determine these MDNs from the secondary URLs and redirect chains recorded by a high-interaction client honeypot. In addition, we propose a novel drive-by download detection method. Instead of depending on the malicious content used by previous methods, our algorithm first identifies and then leverages the *URL*s of the MDN's *central servers*, where a central server is a common server shared by a large percentage of the drive-by download attacks in the same MDN. A set of regular expression-based signatures are then generated based on the URLs of each central server. This method allows additional malicious webpages to be identified which launched but failed to execute a successful drive-by download attack. The new drive-by detection system named ARROW has been implemented, and we provide a large-scale evaluation on the output of a production drive-by detection system. The experimental results demonstrate the effectiveness of our method, where the detection coverage has been boosted by 96% with an extremely low false positive rate.

## Categories and Subject Descriptors

K.6.5 [**Computing Milieux**]: Management of Computing and Information Systems-Security and Protection

## General Terms

Security

## Keywords

Drive-by download, malware distribution network, signature generation, detection

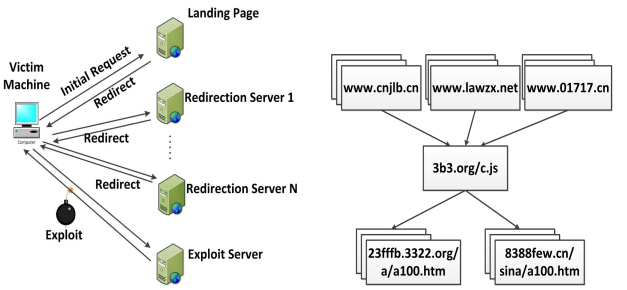
## 1 Introduction

As more people use the Internet for entertainment, commerce, and communication, the web is attracting an increasing number of attacks. In particular, drive-by download attacks are one of the most significant and popular threats on the Internet [12] since they do not usually require user interaction. In drive-by download attacks, attackers embed malicious content in either the original webpage visited by the user, denoted as the landing page, or some content directly, or indirectly, referenced by the landing page, which is usually hosted on a compromised web server. When a browser renders these webpages, the malicious content attempts to exploit a browser vulnerability. A successful exploit attempt often causes malware to be downloaded allowing the attacker to control the underlying operating system for various malicious activities (e.g., information stealing, denial of service attacks, etc.).

Multiple steps are usually involved in a drive-by download attack, as illustrated in Figure 1(a). The malicious content embedded in the compromised webpage, which is initially rendered by the browser, usually does not directly exploit the browser. Instead it redirects the browser to other redirection servers which either provide external webpage content or further redirect to additional servers. After visiting one or more, possibly benign, redirection servers, the browser will eventually encounter a malicious redirection server which further redirects to the servers that attempt to exploit the browser and download malware. The malicious redirection server can be used to manage the attacks and decide the exploit server to use, which has the best matching set of exploits (e.g., IE exploits for IE browsers). A set of different drive-by downloads can be managed by the same attackers for a particular purpose (e.g., distributing the same malware binary for a botnet) and form a *malware distribution network* (MDN). In this paper, we define a MDN to be a collection of drive-by downloads that serve the same malicious objective such as distributing related malware executables.

Several methods [2, 6, 8, 9, 10, 20] have been proposed to detect drive-by download attacks and are described in detail in Section 2. Most of these methods [2, 6, 9, 10, 20] depend on the malicious *webpage content* returned by *servers used for exploits or malware distribution* to detect the attacks. For example, [2, 6, 10, 20] require the exploit content whereas [9] needs the downloaded binary. These systems may fail to detect a large number of drive-by download attacks (i.e. false negatives) if no exploit is detected for several reasons. The attackers can be aware of IP addresses of the detection systems. In this case, they can feed the browsers with benign content. Additionally, the detec-


*WWW 2011*, March 28–April 1, 2011, Hyderabad, India.
ACM 978-1-4503-0632-4/11/03.




(a) Drive-By Download (b) MDN with Central Server

**Figure 1:** Examples

tors may not be configured correctly to match an attack. For example, the detector may present an unpatched version of Internet Explorer, but the malicious landing page may target a FireFox vulnerability. Finally, since the exploit and malware distribution servers may be hosted on compromised servers, their stability will be affected by the network churn. So when the browser visits these servers, they may be temporally unavailable and thus no exploit attempt occurs in the victim's browser. Although not capable of detecting drive-by download attacks, `WebCop` [8] (see Section 2 for details), can handle this problem since it is based on the URLs of exploit/malware-distribution servers. However, the hostnames, IP addresses or the parameter values in the URLs can be frequently changed to make `WebCop` ineffective since it matches the entire URL of the malicious executable.

In this paper, we propose a novel detection method, described in Section 3, by leveraging the URL information of the *central servers* in MDNs, where a central server is a common server shared by a large percentage of the drive-by download samples in the same MDN. An example of an MDN with a central server is presented in Figure 1(b), where "3b3.org" serves as a central server. A central server usually provides some critical information to make the drive-by download successful, and it is not necessarily the server used for exploit attempts or malware distribution. For example, it can be a redirection server used to optimize the MDN performance. A central server can even be a legitimate server where certain information is retrieved to calculate the location of the exploit servers dynamically, as presented in Section 4.3. To be specific, our method bootstraps from the drive-by download samples detected using existing methods, where we first aggregate drive-by download samples into MDNs based on the malware (i.e., hash value) information or the URL of the exploit server. For each MDN, we next discover the central servers if they exist. We further generate signatures in the form of regular expressions based on the URLs for the central servers. These signatures can then be distributed to a search engine or browsers to detect drive-by downloads. The lower half of Figure 2 illustrates our method. These signatures can boost the detection coverage in three ways. First, if a drive-by download attempt reaches the central server without hitting the servers for exploit attempts or malware distribution, our signatures can still detect the attack. Second for a drive-by download attempt, if there is only a URL request to the central server without malicious webpage content returned, our signatures can still detect it since the signatures are independent of the webpage content. Third, the signatures are in the form

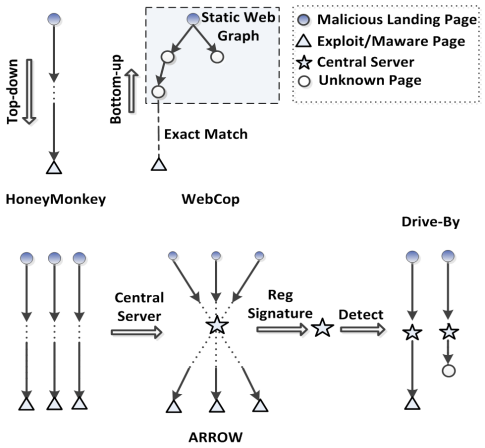



**Figure 2:** Drive-by Download Detection Methods

of regular expressions, which can capture the structural patterns of a central server's URL and therefore outperform exact string matching used by `WebCop`. It is noteworthy that we do not intend to use our method to replace the current systems; we view it as a complementary way to boost the coverage of existing drive-by download detectors.

We have implemented our method in a system named `ARROW` and validated it using data generated from a large-scale production search engine. The experimental results in Section 4 demonstrate that our method can significantly increase the detection coverage by 96%, with an extremely low false positive rate. A discussion of the system is provided in Section 5. In summary, the paper includes the following contributions:

1. We provide a method to identify malware distribution networks from millions of individual drive-by download attacks.
2. By *correlating* drive-by download samples, we propose a novel method to generate regular expression signatures of central servers of MDNs to detect drive-by downloads.
3. We build a system called `ARROW` to automatically generate regular expression signatures of central servers of MDNs and evaluate the effectiveness of these signatures.

## 2 Related Work

Researchers have made great efforts to analyze and detect web-based malware attacks. These existing methods, summarized in Table 1, can be categorized into 2 classes, namely top-down and bottom-up. The table also illustrates how `ARROW` compares to prior work based on other features including whether the detection examines the content or the URL of the webpage, uses the results of the static or dynamic crawler, and correlates multiple instances of an attack.

The top-down approach for drive-by detection adopts a crawler-scanner based architecture. For drive-by downloads, the crawler collects URLs by traversing the dynamic web graphs in the forward direction, while in parallel, a scanner identifies drive-by download attempts. The scanner could be a client honeypot using signature [4] and anomaly detection [3] methods. By rendering a URL, the scanner investigates the suspicious state change of the operating system

| Approach | Top-Down | Bottom-Up | Page Content | URL | static crawler | dynamic crawler | Correlation of multiple Drive-by Downloads |
|---|---|---|---|---|---|---|---|
| HoneyMonkey [20] | ✓ | | ✓ | | | ✓ | |
| Crawler-Based Spyware Study [2] | ✓ | | ✓ | | ✓ | ✓ | |
| Capture-HPC [3, 5] | ✓ | | ✓ | | | ✓ | |
| IFrame [12] | ✓ | | ✓ | | | ✓ | |
| PhoneyC [6] | ✓ | | ✓ | | | ✓ | |
| Malicious JavaScript Detection [10] | ✓ | | ✓ | | | ✓ | ✓ |
| Blade [9] | ✓ | | ✓ | | | ✓ | |
| WebCop [8] | | ✓ | | ✓ | ✓ | | |
| ARROW | | ✓ | | ✓ | | ✓ | ✓ |

**Table 1:** Comparison of Different Detection Methods

or analyzes the webpage to detect drive-by downloads. The upper-left part of Figure 2 presents the architecture of `HoneyMonkey` as an example for the top-down approach. Most of the drive-by download detection approaches [2, 5, 6, 9, 10, 12, 20] fall in the top-down category. For example, a high-interaction client honeypot scanner [3] has been used to dynamically execute the webpage content [2, 5, 20]. Provos et al. [12] also adopted high-interaction client honeypots to conduct large-scale measurements of drive-by downloads in the wild. Nazario [6] proposed a lightweight, low-interaction client honeypot called `PhoneyC` to detect drive-by downloads by analyzing the webpage content. Cova et al. [10] developed a machine learning-based detector to investigate the JavaScript embedded in webpage content for drive-by download detection. Recently, Lu et al. [9] designed a detector to identify drive-by downloads by correlating user action and the binary downloading events. These approaches have shown promising results. However, their effectiveness is significantly limited by the availability of a successful response with malicious content from a drive-by download attack. The lack of a malicious response will make these approaches ineffective and thus may introduce a large number of false negatives.

To deal with the limitations introduced by analyzing webpage content, Stokes et al. [8] proposed a bottom-up based approach, called `WebCop`, to identify webpages that host malicious binaries. `WebCop` takes two inputs, i) a set of URLs for malware distribution sites that are contributed by a large number of anti-malware clients, and ii) a static web graph. `WebCop` traverses the hyperlinks in the web graph in a reverse direction to discover the malicious landing pages linking to the URLs for malware distribution sites. In Figure 2, the upper-middle part illustrates the architecture of `WebCop`. Nevertheless, `WebCop` has two limitations. First, `WebCop` uses an *exact* match to discover the malware distribution sites in the web graph, which may easily introduce false negatives especially when the parameter values are changed. Second, the web graph is based on static hyperlinks, which limit the detection of `WebCop`. For example, a malicious landing page may redirect the browser to the exploit server only if its dynamic content (e.g., malicious JavaScript code) is executed in the browser. Therefore, a static web graph has very limited visibility of the route from malicious landing pages to the malware distribution sites.

Provos et al. [12] proposed a method to discover malware distribution networks from drive-by download samples. This method requires the parsing and matching operation of the webpage headers (e.g., `Referer`) and content (e.g., JavaScript), which is heavyweight. Furthermore, the attackers could potentially obfuscate the webpage content to prevent their MDNs from being discovered. In contrast, `ARROW` identifies MDNs by aggregating drive-by download samples into different groups according to the malware hash values and URL information of exploit servers. Although this method provides less information of MDNs for measurement purpose (e.g., the number of redirection steps) compared to [12], it provides enough information for `ARROW` to detect central servers. In particular, this approach is more efficient and robust, which can identify more MDNs given a large number of drive-by download samples.

Automatic signature generation based on network information has been studied in previous work [7, 14, 15, 19, 21, 22] and has been used to detect various attacks. For example, most [7, 15, 19, 22] focus on *worm fingerprinting*. Perdisci et al. [14] generate signatures to detect *HTTP-based malware* (e.g., bots). `AutoRE` [21] outputs regular expression signatures for *spam detection*. Compared to these methods, `ARROW` mainly differentiates itself by detecting a different attack (a.k.a, drive-by download). Although `ARROW` uses a similar approach to build keyword-based signature trees proposed by Xie et al. [21], we have adapted it to drive-by download detection since drive-by downloads have different characteristics compared to those of spam (e.g., "distributed" and "bursty"), which are critical to guide `AutoRE` to build the signature tree.

# 3 System

Figure 3 presents the architecture of the `ARROW` system. The input of the system is a set of HTTPTraces, which will be described in the following sections, and the output is a set of regular expression signatures identifying central servers of MDNs. There are three processing components in the system. The first component, Hostname-IP Mapping, discovers groups of hostnames closely related by sharing a large percentage of IP addresses. The second component, central server identification, aggregates individual drive-by download samples which form MDNs and then identifies the central servers. For an MDN with one or more central servers, the third component generates regular expression signatures based on the URLs and also conducts signature pruning. The following sections describe the system's input, processing components, and output in detail.

## 3.1 HTTPTraces

**HTTPTrace**s are initially collected from a cluster of high-interaction client honeypots, and an HTTPTrace example is presented in Table 2. Each honeypot visits the URL of the landing page and executes all of the dynamic content (e.g., JavaScript). By detecting state changes in the browser and the underlying operating system and file system, a honeypot can identify whether the landing page is suspicious. The suspicious landing page and other URLs consequently visited are recorded. If any exploit content is detected on


**Figure 3:** System Architecture

| HTTPTrace | |
|---|---|
| Landing Page | www.foo.com/index.php |
| URLs | www.bar.com/go.js<br>www.redirect.com/rs.php<br>www.exploit.com/hack.js<br>www.malware.com/binary.exe |
| IPs | www.bar.com - 192.168.1.2<br>www.redirect.com - 192.168.1.3<br>www.exploit.com - 192.168.1.4<br>www.malware.com - 192.168.1.5 |
| exploitURLs | www.exploit.com/hack.js<br>www.malware.com/binary.exe |
| bHASH | 4A19D50CBBBB702238....358 |
| isDriveBySucc | True |

**Table 2:** An Example of an HTTPTrace

a webpage, the crawler will also record its corresponding URL in "exploitURLs". Otherwise "exploitURLs" is set to be empty. Simply visiting a webpage with no user interaction should never cause an executable to be written to the file system. If the high-interaction client honeypot detects that an executable is written to disk, the file's hash value (e.g. SHA1) is stored in "bHash". The IP address corresponding to the hostname involved in each URL is also recorded. Either the non-empty "exploitURLs" or the non-empty "bHash" implies that a drive-by download attack has been *successfully launched and detected*, where "isDriveBySucc" is set as `TRUE`. Otherwise, "isDriveBySucc" is set as `FALSE`. It should be noted that "isDriveBySucc=`FALSE`" may indicate a false negative. For example, the exploit content is hosted in a compromised server and is temporally unavailable at the moment of detection. Although "isDriveBySucc" is set to be `FALSE` in this case, this exploit webpage is still considered to be harmful to other users.

## 3.2 Hostname-IP Mapping

We typically use a hostname or an IP address to represent a server. However, attackers can introduce great diversity of hostnames and IP addresses for an individual server. On one hand, IP addresses may exhibit great diversity due to fast-flux techniques [16, 17], where one hostname can be resolved to a large number of IP addresses. In this case, using the IP address to represent one server decreases the possibility of identifying central servers that share the same hostname but distribute to different IP addresses. On the other hand, attackers can register a number of hostnames (and thus domain names) and resolve them to one or a small pool of IP addresses. In this case, if we use a hostname to represent one server, we may fail to identify central servers with the same IP address but different hostnames.

In order to eliminate the diversity introduced by hostnames and IP addresses for representing a server, we design a data structure named **Hostname-IP Cluster** ($HIC$) to represent a group of hostnames that share a large percentage of IPs. A similar technique [13] was proposed to discover fast-flux networks. Each $HIC$ has a set of hostnames and a set of IP addresses, denoted as $HIC = \{S_{Host}, S_{IP}\}$. We follow the steps below to discover $HIC$s:

1 As the initialization phase, for each hostname ($h_i$) we have observed in the HTTPTraces, we identify all of the IP addresses resolved for $h_i$ $(IP_1, IP_2, \ldots, IP_n)$. We initiate one $HIC_i$ using $HIC_i.S_{Host} = \{h_i\}$ and $HIC_i.S_{IP} = \{IP_1, \ldots, IP_n\}$.

2 Suppose we have $N$ Hostname-IP clusters, denoted as $HIC_1, \ldots, HIC_N$. For each pair of $HIC_i$ and $HIC_j$, we investigate $r_{HIC} = \frac{|HIC_i.S_{IP} \cap HIC_j.S_{IP}|}{|HIC_i.S_{IP} \cup HIC_j.S_{IP}|}$, the overlap of the IP addresses of the two clusters. If $r_{HIC} > T_{HIC}$, where $T_{HIC}$ is a pre-defined threshold and $0 \leq T_{HIC} < 1$, we first merge the second cluster into the first cluster using $HIC_i.S_{IP} = HIC_i.S_{IP} \cup HIC_j.S_{IP}$ and $HIC_i.S_{Host} = HIC_i.S_{Host} \cup HIC_j.S_{Host}$ and then discard $HIC_j$.

3 Repeat step 2 until no $HIC$s are merged into other clusters.

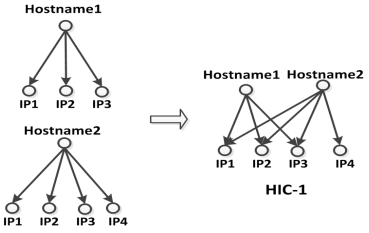



**Figure 4:** Hostname-IP Mapping

$T_{HIC}$ is a pre-defined threshold to determine whether two $HIC$s should be merged or not. A small value for $T_{HIC}$ indicates a relaxed condition. For example, $T_{HIC} = 0$ means that if two $HIC$s share a single common IP, they are going to be merged together. This may introduce significant noise in the merged $HIC$s. A large value of $T_{HIC}$ enforces a strong condition for two $HIC$s to be merged. For example, a $T_{HIC}$ value close to 1 requires that two $HIC$s share almost the same IPs. This may significantly decrease the possibility of merging more related $HIC$s. In the current system, we set $T_{HIC} = 0.5$.

Figure 4 gives an example of identifying Hostname-IP clusters, where `Hostname1` and `Hostname2` share 3 out of 4 IP addresses ($r_{HIC} = 75\%$) and are grouped in one cluster. After identifying all the $HIC$s, we use the index of each $HIC$ to replace the hostname in each URL. For instance as described in Figure 4, the URLs of `hostname1/t.php?s=1` and `hostname2/t.php?s=2` will be represented as `HIC1/t.php?s=1` and `HIC1/t.php?s=2`. Therefore, instead of taking `hostname1` and `hostname2` as two different servers, we use the $HIC$ to discover their relationship and represent them as a single server.

## 3.3 Identification of Central Servers

To identify the MDNs with one or more central servers, we need to first discover MDNs from a set of HTTPTraces. In `ARROW`, we use the hash value of the malware executable ("bHash") and URLs of exploit webpages ("exploitURLs") to aggregate HTTPTraces into MDNs. On one hand, we

group all HTTPTraces with the same value of "bHash" into one MDN. It is possible that multiple organizations may install the same malware causing the two groups' MDNs to be merged. This should not be a major problem for the following reason. For malware that is automatically generated from a toolkit, a malicious executable is often customized for each attacker yielding different executable hash values: these similar attacks will be grouped into separate MDNs. On the other hand, for each URL in "exploitURLs", we identify its corresponding $HIC$ and group all the HTTPTraces with same $HIC$ index into one MDN. For each MDN, we do not consider the landing pages as candidate URLs for discovering central servers, since they are usually compromised websites that are unlikely to serve as central servers. Also, we do not consider the URLs of "exploitURL" as candidate URLs; therefore, the identified central servers are likely used for redirection purpose.

After we aggregate all of the HTTPTraces into different MDNs, we eliminate (i.e. filter) the MDNs with a small number of HTTPTraces since small MDNs have a higher likelihood of incorrectly identifying central servers. For example, an MDN with two HTTPTraces may have a high probability of belonging to the same advertisement network, and thus the benign, advertisement server will be incorrectly identified as a central server in the MDN. Therefore `ARROW` only identifies MDNs containing more than $T_{HTTPTrace}$ drive-by download samples where $T_{HTTPTrace}$ is currently set as $T_{HTTPTrace} = 20$.

Then for each MDN, we identify the central servers as the nodes (represented by an $HIC$ index) that are contained in a majority of the HTTPTraces. Given an MDN with $K$ HTTPTraces where each trace contains a collection of URLs, we have first replaced the hostname for each URL using its corresponding $HIC$. For each $HIC$, we determine the count, $C$, of the number of HTTPTraces employing this $HIC$. If $r_{cen} = \frac{C}{K}$ is greater than the pre-defined ratio $T_{cen}$, where $0 \leq T_{cen} \leq 1$, we take this $HIC$ as a central server. We conservatively set $T_{cen}$ with a large value (currently $T_{cen} = 0.9$) to guarantee the "central" property of the central server. For any two MDNs sharing one or more central servers, we merge them together into a new MDN. The shared central servers are taken as the central servers for the new MDNs, while the other central servers are discarded. This operation eliminates redundant central servers without compromising their coverage, and thus reduces the total number of signatures and consequently computationally expensive, regular expression matching operations. Also by merging smaller MDNs, we increase the number of URLs corresponding to each central server, which helps to generate more generic signatures. Figure 5 illustrates an example of discovering MDNs and central servers. In Figure 5, $S_1$, $S_2$ and $S_3$ are initially identified as central servers since most of the HTTPTraces corresponding to "$Malware_{1/2/3}$" ($bHash_{1/2/3}$) contain $S_{1/2/3}$. The central server $S_1$, which is shared by two MDNs, is ultimately identified as the central server for the newly merged MDN. Although $S_2$ is discarded, $S_1$ still guarantees its coverage of drive-by download samples in this MDN.

## 3.4 Regular Expression Generation

In this section, we discuss how to generate regular expressions corresponding to the central servers in order to detect additional HTTPTraces exhibiting drive-by download

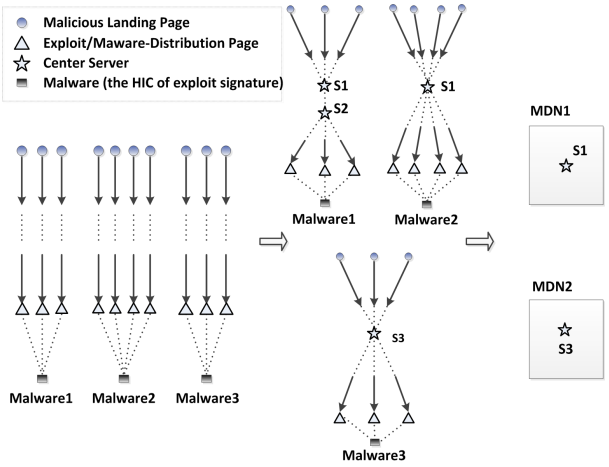



**Figure 5:** Discover MDNs and Identify Central Servers

attempts. There are two straight forward approaches to using the central server information to detect HTTPTraces of drive-by downloads. One option is to use the network level information of central servers (i.e. hostname and IP address). For example, if any URL in an HTTPTrace contains the hostname or IP address of any central server, this HTTPTrace will be labeled as suspicious. However, this detection approach is too coarse and may introduce a large number of false positives, especially when the central server is benign such as the example described in Section 4.3. Another option is to use exact string matching of URLs in the MDNs corresponding to central servers. For example, if any URL in an HTTPTrace exactly matches any of the URLs of central servers, this HTTPTrace will be labeled as suspicious. However, this approach is too specific resulting in false negatives. For example, a simple change in values for the parameters in the URL makes the exact match fail. These examples provide motivation to design generic signatures that can capture the invariant part of the URLs for central servers and also give an accurate description of the dynamic portion of these URLs. Thus `ARROW` generates regular expression signatures for each central server by investigating the structural patterns of its corresponding URLs.

To generate regular expressions, `ARROW` follows two steps:

1. For each central server in a MDN, generate tokens out of all the URLs that are contained in this MDN and corresponding to the central server, and then build the signature tree according to the coverage of each token.
2. Identify the branches with high coverage, and then generate signatures in the form of regular expressions.

These items are discussed in detail in the next two sections.

### 3.4.1 Token and Signature Tree Generation

Since URLs are well-structured strings, `ARROW` generates tokens based on the structure of each URL. `ARROW` collects tokens for the following 5 categories: the HIC index, the name for each directory, the name of the file, the suffix of the file and the name of the parameters. The information from the last four categories indicates the information of the toolkit that attackers use to organize the MDNs. For example, the name of the directory represents the directory or the sub-directory that organizes the scripts. The suffix of a script, which implies the script language, is usually

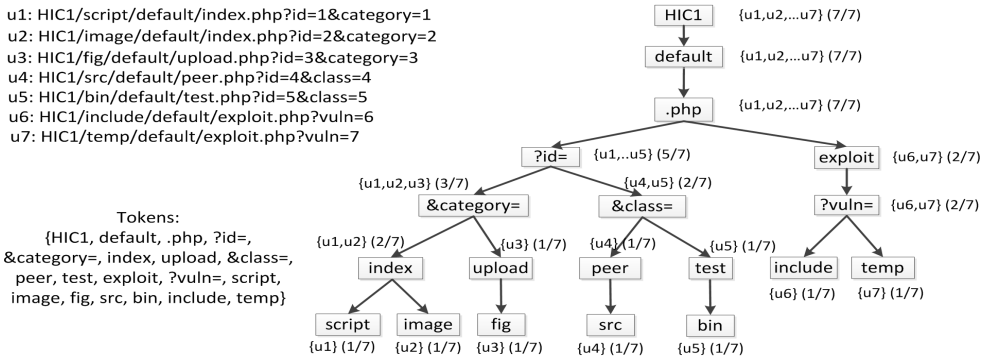



**Figure 6:** An Example of a Signature Tree

identical in the same MDN for a script with the same functionality, even if the file names differ from each other. Taking the 7 URLs in Figure 6 as an example, we find the set of tokens as {`HIC1,default,index,.php,?id=,&category=,upload,&class=,peer,test,exploit,?vuln=,fig,script,image,src,bin,include,temp`}.

We build the signature tree based on the approach introduced in [21]. We denote the set of all URLs as $U_{all}$, and the set of URLs containing one token/node $N_i$ as $U_i^N$. The set of URLs covered by a particular branch $B = \{N_{root}, N_1 \dots N_i\}$ is given by $U_i^B$, where $U_i^B = U_{root}^N \cap U_1^N \cdots \cap U_i^N$. To build the tree, two operations are defined: building the root node and building the child nodes. To build the root node, the token with the largest coverage of the URLs is taken as the root ($N_{root}$). To identify the child node(s) for one node $N_i$ in branch $B = \{N_{root}, N_1 \dots N_i\}$, we follow the two steps below.

1. Let $U_{Rest}^B = U_i^B$ and $Set_{nodes} = \{N_{root}, N_1 \dots N_i\}$.
2. If $U_{Rest}^B = \emptyset$, exit. Otherwise, for each node $N_j \notin Set_{nodes}$, select the one with maximum $|U_{Rest}^B \cap U_j^N|$ as the child node. If $max(|U_{Rest}^B \cap U_j^N|) == 0$, exit. Otherwise, suppose this node is $N_k$, then let $U_{Rest}^B = U_{Rest}^B - U_k^N$ and $Set_{nodes} = Set_{nodes} \cup N_k$. Repeat this step.

Returning to Figure 6, an example of the signature corresponding to the 7 URLs is presented. Since the $HIC$ index, $HIC1$, is always contained in all of the URLs, we take it as the root of the tree.

### 3.4.2 Signature Generation

For each branch in the signature tree, we obtain more specific patterns for a subset of URLs as we approach the leaves. To learn the general pattern representing the URLs, we define a node $N_i$ as a *critical-node* if $|U_i^N|/|U_{all}| \geq R$ and none of its child nodes satisfy this condition where the threshold $0 \leq R \leq 1$. We name the branch from the root to this critical-node as a *critical-branch*. In Algorithm 1 we describe a method to identify each critical-node. We run this algorithm multiple times by decreasing $R$ until all the URLs are covered by the critical-nodes, which is described in Algorithm 2. In our system, we initialize $R = 0.3$ and $\alpha = 0.9$. For the signature tree described in Figure 6, we identify the critical-nodes by following the steps below.

1. $R = 0.3$. Identify critical-node "`&category=`".
2. $R = 0.3 * 0.9$. Identify critical-nodes "`&class=`" and "`?vuln=`".

After identifying the critical-nodes in the signature tree, we traverse the tree to find the branches from the root to each critical-node. The nodes in one of these branches composite one set of candidate tokens. For each token, we further investigate its average distance from the beginning of the URLs. For one set of candidate tokens, we sort the tokens according to the average distance and then obtain a sequence of candidate tokens. For the example above, `ARROW` generates three sequences:

1. $seq_1$ = { `HIC1,default,.php,?id=,&category=` }
2. $seq_2$ = { `HIC1,default,.php,?id=,&class=` }
3. $seq_3$ = { `HIC1,default,.php,exploit,?vuln=` }

Next, for each pair of consecutive tokens (and also for the last token and the end of URLs), we investigate the following properties of the strings that reside between them.

1. Are these strings identical?
2. The minimum and maximum length of the strings.
3. Are the characters in these strings lower or upper case?
4. Are the characters in these string letters or numbers?
5. Enumerate special characters (e.g., "." and "?") in these strings.

Finally, we summarize these properties in order to generate the regular expression. If these strings are identical, we directly present such string in the regular expression. Otherwise, we describe the properties in the regular expression format. For our running example, we obtain the three regular expressions:

1. $reg1$ = `HIC1/[a-z]{3,5}/default/[a-z]{5,6}.php?id=[0-9]{1}&category=[0-9]{1}`
2. $reg2$ = `HIC1/[a-z]{3,3}/default/[a-z]{4}.php?id=[0-9]{1}&class=[0-9]{1}`
3. $reg3$ = `HIC1/[a-z]{4,7}/default/exploit.php?vuln=[0-9]{1}`

We further refer to the hostnames and IP addresses in $HIC1$. We generate the domain names for the hostnames and replace $HIC1$ using the domain names and IP addresses to get the regular expression signatures. For example, if $HIC1 = \{\{cnt1.foo1.com, cnt2.foo1.com, cnt1.foo2.com\}, \{192.168.1.2, 192.168.1.4\}\}$, we replace $HIC1$ using `foo1.com`, `foo2.com`, `192.168.1.2` and `192.168.1.4` for $reg1$, $reg2$ and $reg3$. For example, $reg3$ will be extended to be four signatures:

1. $reg3.1$ = `foo1.com/[a-z]{4,7}/default/exploit.php?vuln=[0-9]{1}`

2. $reg3.2$ = `foo2.com/[a-z]{4,7}/default/exploit.php?vuln=[0-9]{1}`
3. $reg3.3$ = `192.168.1.2/[a-z]{4,7}/default/exploit.php?vuln=[0-9]{1}`
4. $reg3.4$ = `192.168.1.4/[a-z]{4,7}/default/exploit.php?vuln=[0-9]{1}`.

It is possible that some signatures are prone to induce false positives during detection. For example, a signature may be too general corresponding to a legitimate domain name. To decrease the possibility of false positives, we apply signature pruning. To be specific, we evaluate each signature using a large set of legitimate HTTPTraces, where each HTTPTrace is associated with a high-reputation landing page. We discard any signature successfully matching any URL in these legitimate HTTPTraces.

**Algorithm 1** IdentifyCriticalNode($curNode$, $R$, $N$)

```
curNode: one Node in the tree.
R: the threshold (0 < R < 1).
N: N = |U_all|.
numRest: global variable initiated as |U_all|.
begin
    if curNode.isCriticalNode() then
        return;
    Boolean flag = true;
    foreach Node oneNode in curNode.getChildNodes()
    do
        if |U^B_oneNode| / N >= R then
            flag = false;
            break;
    if flag && |U^B_curNode| / N >= R then
        curNode.setCriticalNode();
        numRest = numRest - |U^B_curNode| ;
        return;
    else
        if curNode.isLeaf() then
            return;
        else
            foreach Node oneNode in
            curNode.getChildNodes() do
                IdentifyCriticalNode(oneNode, R, N);
end
```

**Algorithm 2** ExploreTree($R$)

```
R: the threshold (0 < R < 1).
N: N = |U_all|.
numRest: global variable.
α: decreasing ratio.
begin
    numRest = |U_all|;
    while numRest > 0 do
        IdentifyCirticalNode(rootNode, R, N);
        R = R * α;
end
```

# 4 Evaluation

We have implemented a prototype system named ARROW, and evaluated it using a large volume of HTTPTraces. The HTTPTraces, described in Section 3 and Table 2, were collected by evaluating their landing pages using a production cluster of high-interaction client honeypots. The following sections describe the experimental setup and evaluation results.

| Trace | Num |
|---|---|
| $S_{total}$ | 3,500,000,000 |
| $S_{malicious}$ | 1,345,890 |
| $S_{unlabeled}$ | 3,498,654,110 |
| $S_{benign1}$ | 10,811,805 |
| $S_{benign2}$ | 10,733,282 |

**Table 3:** HTTPTraces for Experiments

## 4.1 Experiment Setup

Among all the HTTPTraces produced by the honeypot cluster, we randomly selected a set of 3.5 billion HTTPTraces ($S_{total}$). Out of $S_{total}$, $1,345,890$ HTTPTraces (denoted as $S_{malicious}$) are identified as drive-by download attacks. The remaining HTTPTraces are taken as the unlabeled dataset (denoted as $S_{unlabeled}$). In order to obtain benign HTTPTraces for signature pruning and false positive evaluation, we use the following approach. We first collect a list of 87k hostnames and URLs with high reputation scores that were recently confirmed by the analysts, where a high reputation score for a host indicates an extremely low probability that its webpages are associated with malicious activities including drive-by download attacks, phishing, scamming and spamming. We then randomly divide this list into two sublists and collect HTTPTraces whose landing pages contain one of these hostnames or URLs. The set of HTTPTraces corresponding to the first sublist (denoted as $S_{benign1}$) is used for signature pruning. The remaining HTTPTraces (denoted as $S_{benign2}$), which correspond to the second sublist, are used for false positive evaluation. Table 3 summarizes the number of HTTPTraces included in each data set described above, indicating a large-scale evaluation of the ARROW system.

The ARROW system applies regular expression signatures to match URLs in HTTPTraces. Regular expression matching is naturally computationally expensive. To speed up the matching process, we first aggregate the domain name and IP address associated with each signature into a set. For a URL, only in case its domain name or the corresponding IP address is contained (by a fast hash-based operation) in that set, signature matching is applied. The matching process is further implemented in a large-scale distributed computing infrastructure. Since the signatures will only match on rare occasion, the computational overhead to match ARROW signatures is negligible.

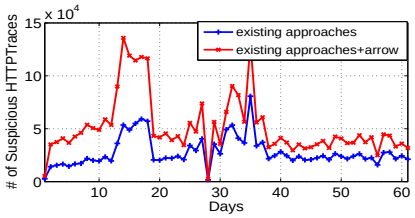
## 4.2 Evaluation Results

We first ran ARROW on the $S_{malicious}$ HTTPTraces to discover Hostname-IP Clusters and identified $14,648$ $HICs$. Some hostnames show strong fast-flux patterns. For example, one $HIC$ has only 6 hostnames but $1,041$ IP addresses, while another $HIC$ has $34,882$ hostnames which resolve to a single IP address. The $HIC$ structure can effectively discover and represent the relationship among such hostnames and IP addresses.

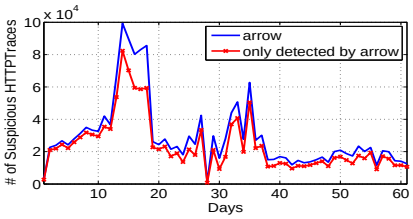
After representing each server with the $HIC$ index, ARROW follows the approach described in Section 3.3 to identify MDNs and central servers. For the HTTPTraces in $S_{malicious}$ that are identified as drive-by download attacks by grouping them based on "bHash" or the $HIC$ index of each URL in "exploitURLs", ARROW identified $6,937$ MDNs in total and 97 MDNs (1.4%) using one or more central servers. By analyzing the URLs for the central servers of these 97 MDNs, ARROW generated $2,592$ regular expression

| Signature |
|---|
| twitter\.com\/trends\/daily\.json\?date\=2[0-9&-]{10,10}callback\=callback2 |
| experimentaltraffic\.com\/cgi\-bin\/009[0-9a-zA-Z?=/.]{4,101} |
| saeghiebeesiogoh\.in\:3129\/js |
| qsfgyee\.com\:3129\/js |
| google\-analitics\.net\/ga\.js\?counter\=[0-9]{1,2} |
| servisesocks5\.com\/el\/viewforum\.php\/[0-9a-z]{32,32}\?spl\=mdac |
| chura\.pl\/rc\/pdf\.php\?spl\=pdf_ie2 |
| trenz\.pl\/rc\/getexe\.php\?spl\=m[a-z_]{3,6} |

**Table 4:** Examples of Signatures

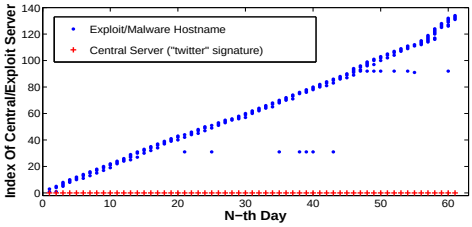



(a) The Existing Method and ARROW

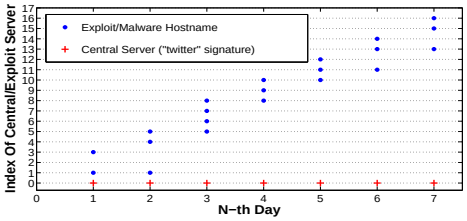



(b) ARROW

**Figure 7:** Detection Results on a Daily Basis




(a) 60 Days




(b) Zoom-In View for First 7 Days

**Figure 8:** Active Days for the Central Server and Exploit Servers

signatures. After pruning these signatures with $S_{benign1}$, ARROW produced 2,588 signatures including the examples presented in Table 4.

We further evaluate the quality of the signatures to answer two questions. i) How many new suspicious HTTPTraces can the signatures contribute? ii) How many false positives are generated by the signatures?

To answer the first question, we apply all the signatures to the 3.5 billion HTTPTraces in $S_{total}$. We name $H_d$ as the set of suspicious HTTPTraces already detected by the high-interaction client honeypots, where $H_d = S_{malicious}$. $H_a$ denotes the set of suspicious HTTPTraces that are detected by the signatures generated by ARROW. Table 5 compares the result of the suspicious HTTPTraces detected by existing honeypots and signatures from ARROW. The column labeled $\frac{H_a \cap H_d}{H_d}$ illustrates that these central server signatures have a coverage of 23.8% of the HTTPTraces detected by existing approaches. The columns of "$H_d$" and "$H_a$" indicate that ARROW signatures identify more suspicious HTTPTraces. In particular, the ARROW signatures contribute a large number of *new* suspicious HTTPTraces (96.0%.) Figures 7(a) and 7(b) present the number of suspicious HTTPTraces detected by ARROW compared to the existing approach on a daily basis. Such boosted detection results demonstrate the significant advantage using ARROW as a parallel drive-by download detection system to existing honeypot-based detection techniques. To answer the second question, we match these signatures against all URLs in $S_{benign2}$. Out of total 10,733,282 legitimate HTTPTraces, only 2 HTTPTraces are matched with the signatures indicating an extremely low false positive rate of 0.0019%.

## 4.3 Case Study for the Twitter Signature

Out of the signatures generated by ARROW, "twitter.com" appears in one signature in Table 4. This signature identified a large number of 135,875 suspicious HTTPTraces as described in column "$H_a$" in Table 6, contributing 8.4% ($\frac{135875}{1612166}$) of all detected suspicious HTTPTraces. Since "twitter.com" is a well known website used for social networking and microblogging, false positives are a concern. In this section, we conduct a detailed analysis of all HTTPTraces matched by the "twitter.com" signature to assess whether or not it is a false positive.

First, we briefly describe why twitter was involved in a large number of drive-by download pages. Manual analysis reveals that a suspicious webpage retrieves the week's top trending terms using Twitter's API, dynamically constructs a hostname from these changing terms, and then instructs the browser to retrieve the exploit from the server corresponding to that hostname. A detailed description of the script that performs these actions can be obtained from a website dedicated to raising awareness of online threats [1].

Figure 8(a) and its zoom-in view Figure 8(b) present the temporal patterns of the MDN identified by the "twitter.com" central server. The X-axis illustrates the days the hostname appears in our collected HTTPTraces over time, whereas the Y-axis represents the index into the set of dynamically generated hostnames based on Twitter's API. As the graph illustrates, the hostname is switched on a regular basis representing a strong *fast-flux* pattern, while the central server ("twitter.com") remains *stable*. This architecture introduces a great challenge for the detection techniques that identify

| Metric | $\lvert H_d \rvert$ | $\lvert H_a \rvert$ | $\lvert H_a \cap H_d \rvert$ | $\lvert H_a - H_d \rvert$ | $\frac{\lvert H_a \cap H_d \rvert}{\lvert H_d \rvert}$ | $\frac{\lvert H_a - H_d \rvert}{\lvert H_d \rvert}$ |
|---|---|---|---|---|---|---|
| Value | 1,345,890 | 1,612,166 | 320,310 | 1,291,856 | 23.8% | 96.0% |

**Table 5:** Evaluation Results

| Metric | $\lvert H_a \rvert$ | $\lvert H_d \rvert$ | $\frac{\lvert H_a - H_d \rvert}{\lvert H_d \rvert}$ | $\lvert H_k \rvert$ | $ReflectRate(H_k, H_a)$ |
|---|---|---|---|---|---|
| Value | 135,875 | 60,159 | 125.9% | 119,774 | 99.6% |

**Table 6:** Evaluation Results For "Twitter" Signature

the server responsible for hosting the actual exploit. In this MDN, this server changes every few days. However, the signature generated by `ARROW` captures the central server, which is the most stable point over time. Therefore, signatures generated by `ARROW` can detect a large number of the suspicious HTTPTraces, even if the corresponding server hosting the exploit changes or is temporarily unavailable.

To assess the false positive rate of the signature, we need to further categorize the matched HTTPTraces. The column $H_a$ is described previously. In addition, 60,159 HTTPTraces, denoted as $H_d$ in Table 6, in the $H_a$ traces have been identified as drive-by downloads by the existing approach. These HTTPTraces in $H_a$ can be classified as follows into three categories:

1. The client honeypot retrieves the exploit from the aforementioned server and is compromised. The client honeypot can successfully detect the drive-by downloads in this scenario.
2. The client honeypot makes a request to the exploit server but fails to retrieve the exploit.
3. The client honeypot visits "twitter.com", but no further connection attempts are made to the exploit server.

Manual analysis reveals that an HTTP request to the server hosting the exploit contains specific keywords as part of the URL's path. These keywords are listed in Table 7 and would be an indication of malicious intent (categories 1 and 2 above). In total, 119,774 or 88.15% of HTTPTraces match these keywords as described in column "$H_k$" in Table 6 leaving a large portion of 11.85% as potential false positives. Since no subsequent requests are made after "twitter.com" has been contacted for category 3, it is challenging to obtain evidence on whether these HTTPTraces also have malicious intent. However, if we assume that a compromised server hosts a large number of similar malicious webpages, we can further assess the malicious intent of these HTTPTraces. We do so with a function called *ReflectRate*, which expresses the rate of suspicious HTTPTraces based on that assumption.

| keywords | "*/tre/*", "*/nte/*", "*/ld/*", "*.exe*" |
|---|---|

**Table 7:** Keywords for Detection

The *ReflectRate* is calculated as follows. We use $ht$ to represent an HTTPTrace and $H$ to denote a set of HTTPTraces ($H = \{ht_1, ht_2, \ldots, ht_n\}$). We further use the functions $LP()$ to return the landing page for an HTTPTrace and $Host()$ to obtain the hostname of a URL. Given $H$, we introduce a function $HostSet()$ to indicate the set of hostnames in the landing pages, $HostSet(H) = \cup_{i=1}^{n}\{Host(LP(ht_i))\}$. We then define a function $Reflect(H_1, H_2)$ to return a set of HTTPTraces $H_r = Reflect(H_1, H_2)$, where $H_r = \{ht_i | ht_i \in H_2, Host(LP(ht_i)) \in HostSet(H_1)\}$. $H_r$ represents a subset of HTTPTraces in $H_2$, whose landing pages are hosted on servers (e.g., $HostSet(H_1)$) that have been confirmed to serve malicious webpage content. Based on these functions, we define $ReflectRate(H_1, H_2) = \frac{|H_r|}{|H_2|}$. For example, suppose $H_1 = \{ht_1, ht_2\}$ is a set of suspicious HTTPTraces detected by existing methods, where $LP(ht_1) =$ `www.foo.com/index.php` and $LP(ht_2) =$ `www.bar.com/index.php`. We also find $HostSet(H_1) = \{$ `www.foo.com`, `www.bar.com`$\}$. Assume we are also presented with a set of unlabeled HTTPTraces $H_2 = \{ht_a, ht_b, ht_c, ht_d\}$, where $LP(ht_a) =$ `www.foo.com/contact.php`, $LP(ht_b) =$ `www.foo.com/test.php`, $LP(ht_c) =$ `www.bar.com/temp.php` and $LP(ht_d) =$ `www.test.com/index.php`. We can classify three out of four HTTPTraces in $H_2$ as suspicious, which is $Reflect(H_1, H_2) = \{ht_a, ht_b, ht_c\}$, since $Host(LP(ht_{a/b/c})) \in HostSet(H_1)$. The resulting $ReflectRate(H_1, H_2)$ would be 75%.

Applied to our case study of "twitter.com", we obtain a *ReflectRate* of 99.6% leaving 0.4% or 544 of the identified traces as potential false positives as shown in the last column of Table 6.

## 5 Discussion

Our evaluation has shown that the `ARROW` system successfully boosts detection of suspicious pages with an overall low false positive rate of 0.0019% and a false positive rate of 0.4% on difficult cases in which the signature points to a legitimate server, such as "twitter.com". Considering the number of webpages inspected by a search engine, even those low false positive rates result in a large absolute number of false positives. If we apply the overall false positive rate of 0.0019% to our sample of 3.5 billion webpages, running `ARROW` will result in approximately 65,000 false positives. The ability to run `ARROW` in production may be compromised if the signatures cause popular websites to be detected as malicious (i.e. false positives). Strategies to reduce the likelihood of this happening have been discussed, such as the *ReflectRate* introduced in the previous section. Alternative strategies to avoid false positives, such as white listing or clustering based on page characteristics, are left for future work.

Signature pruning as well as our evaluation of the overall false positive rate is based on a large data set of benign HTTPTraces. Collecting a comprehensive dataset of definite benign traces, however, is very hard in practice. One could collect URLs corresponding to the most popular queries in the search engine logs or URLs that are most popular in the ISP or enterprise networks with the assumption being that popular webpages are less likely to host drive-by downloads. Nevertheless, as attackers usually compromise legitimate sites to hijack its traffic as an entry point into their MDN, this assumption may not always hold true. It may result in some signatures being pruned that shouldn't have been and some HTTPTraces being incorrectly marked as false positives. However, given that malicious webpages are overall quite rare among popular webpages (e.g., 0.6% of the top million URLs in Google's search results led to malicious

activity according to [12]), the vast majority of HTTPTraces collected in such a way will indeed be benign and can be used for pruning and evaluation.

Evasion may be another concern with our system. Similar to other drive-by download detection systems, by knowing our detection algorithm, attackers can always carefully redesign the attack strategy to evade our detection. For example, the attackers can generate a different binary for each time of an attack, which will in consequence cause different binary hash values and thus prevent `ARROW` from aggregating them into the same MDN. To deal with this problem, more information, such as the similarity of different binaries or behavior characteristics [11, 14, 18], can be adopted to aggregate polymorphic malware into MDNs. The attackers can also decentralize the MDNs to eliminate the central servers or hide the MDN structure from the client (e.g. through server side includes instead of redirect chains.) As the MDNs identified by `ARROW` do not provide comprehensive coverage on all HTTPTraces (with current coverage 23.8%) identified by traditional methods, it is an indication that some of these evasion techniques are used today. These are accepted shortcomings of our approach, and we will look towards refining our existing algorithm and exploring new approaches to detect MDNs as part of our future work.

## 6 Conclusion

Detecting drive-by download attacks is an extremely important and challenging problem. We have shown that by aggregating data from a large number of drive-by download attempts, we are able to discover both simple malware distribution networks as well as more complex MDNs including one or more central servers. We have conservatively estimated that 1.4% (97 out of 6,937) of MDNs employ central servers. While this percentage is currently small, the overall coverage of these complex MDNs is 23.8% (320,310 out of 1,345,890 malicious traces) which is reasonably large considering the small number of actual MDNs with central servers. Going forward, we expect the number of attackers employing sophisticated methods to manage their operations to grow.

The major hurdle to deploying `ARROW` to detect MDNs utilizing central servers and block all landing pages which redirect to these servers is developing ways to accurately identify false positives. This problem is generic to any method attempting to solve this problem and is not a reflection of the proposed system. We have shown that the regular expression signatures have a very low false positive rate when compared to a large number of high reputation sites. We have also manually investigated many of the signatures and found that they appear to be malicious. However, it is not practical to employ an army of analysts to investigate all signatures generated by the system, particularly given the highly dynamic ecosystem being used by attackers today. Developing better, automated methods of assessing the purpose of these central servers in the absence of a successful attack needs to be a focus of future research. Without access to this ideal system in the near term, it may be prudent to restrict internet content from users (i.e. not display the webpages from a search query or serve ads which redirect to the central server) in order to err on the side of caution. While this may potentially penalize legitimate content providers initially as the system is deployed, having a method for an individual organization to understand the underlying cause of any true false positives and a method to quickly rectify any errors will help balance the competing objectives of providing users the widest range of content while keeping them safe from harm.

## 7 References


[1] Hackers use twitter api to trigger malicious scripts. http://blog.unmaskparasites.com/2009/11/11/hackers-use-twitter-api-to-trigger-malicious-scripts/, 2009.

[2] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware on the web. In *Proc. NDSS*, 2006.

[3] C. Seifert and R. Steenson. Capture - honeypot client (capture-hpc). https://projects.honeynet.org/capture-hpc, 2006.

[4] C. Seifert, I. Welch and P. Komisarczuk. Honeyc - the low-interaction client honeypot. In *Proc. NZCSRCS*, 2007.

[5] C. Seifert, R. Steenson, T. Holz, B. Yuan and M. A. Davis. Know your enemy: Malicious web servers. http://www.honeynet.org/papers/mws/, 2007.

[6] J. Nazario. Phoneyc: A virtual client honeypot. In *Proc. LEET*, 2009.

[7] J. Newsome, B. Karp and D. Song. Polygraph: automatically generating signatures for polymorphic worms. In *Proc. IEEE Symposium on Security and Privacy*, 2005.

[8] J. W. Stokes, R. Andersen, C. Seifert and K. Chellapilla. Webcop: Locating neighborhoods of malware on the web. In *Proc. USENIX LEET*, 2010.

[9] L. Lu, V. Yegneswaran, P. Porras and W. Lee. Blade: An attack-agnostic approach for preventing drive-by malware infections. In *Proc. ACM CCS*, 2010.

[10] M. Cova, C. Kruegel and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proc. WWW*, 2010.

[11] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian and Jose Nazario. Automated classification and analysis of internet malware. In *Proc. RAID*, 2007.

[12] N. Provos, P. Mavrommatis, M. Abu Rajab and F. Monrose. All your iframes points to us. In *Proc. USENIX SECURITY*, 2008.

[13] R. Perdisci, I. Corona, D. Dagon and W. Lee. Detecting malicious flux service networks through passive analysis of recursive dns traces. In *Proc. ACSAC*, 2009.

[14] R. Perdisci, W. Lee and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proc. NSDI*, 2010.

[15] S. Singh, C. Estan, G. Varghese and S. Savage. Automated worm fingerprinting. In *Proc. USENIX OSDI*, 2004.

[16] T. Holz, C. Gorecki, K. Rieck, F. C. Freiling. Measuring and detecting fast-flux service networks. In *Proc. NDSS*, 2008.

[17] The Honeynet Project. Know your enemy: Fast-flux service networks; an ever changing enemy. http://www.honeynet.org/papers/ff/, 2007.

[18] U. Bayer, P. Milani, C. Hlauschek, C. Kruegel and E. Kirda. Scalable, behavior-based malware clustering. In *Proc. NDSS*, 2009.

[19] V. Yegneswaran, J. T. Giffin, P. Barford and S. Jha. An architecture for generating semantics-aware signatures. In *Proc. USENIX SECURITY*, 2005.

[20] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen and S. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proc. NDSS*, 2006.

[21] Y. Xie, F. Yu, K. Achan, R. Panigraphy, G. Hulten and I. Osipkov. Spamming botnets: Signatures and characteristics. In *Proc. ACM SIGCOMM*, 2008.

[22] Z. Li, M. Sanghi, B. Chavez, Y. Chen and M. Kao. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proc. IEEE Symposium on Security and Privacy*, 2006.