Delta: Automatic Identification of Unknown Web-based Infection Campaigns

Kevin Borgolte, Christopher Kruegel, Giovanni Vigna
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, California, United States of America
kevinbo,chris,vigna@cs.ucsb.edu

ABSTRACT

Identifying malicious web sites has become a major challenge in today's Internet. Previous work focused on detecting if a web site is malicious by dynamically executing JavaScript in instrumented environments or by rendering web sites in client honeypots. Both techniques bear a significant evaluation overhead, since the analysis can take up to tens of seconds or even minutes per sample.

In this paper, we introduce a novel, purely static analysis approach, the Δ-system, that (i) extracts change-related features between two versions of the same website, (ii) uses a machine-learning algorithm to derive a model of web site changes, (iii) detects if a change was malicious or benign, (iv) identifies the underlying infection vector campaign based on clustering, and (iv) generates an identifying signature.

We demonstrate the effectiveness of the Δ-system by evaluating it on a dataset of over 26 million pairs of web sites by running next to a web crawler for a period of four months. Over this time span, the Δ-system successfully identified previously unknown infection campaigns. Including a campaign that targeted installations of the Discuz!X Internet forum software by injecting infection vectors into these forums and redirecting forum readers to an installation of the Cool Exploit Kit.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; D.4.6 [**Software Engineering**]: Security and Protection—*Invasive software (malware)*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Clustering, Information filtering, Selection process*

Keywords

computer security; web-based malware; malware detection; infection vector identification; infection campaigns; clustering; trend detection; web dynamics

CCS'13, November 4–8, 2013, Berlin, Germany.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516725.

1 INTRODUCTION

The rapid growth and widespread access to the Internet, and the ubiquity of web-based services make it easy to communicate and interact globally. However, the software used to implement the functionality of web sites is often vulnerable to different attack vectors, such as cross-site scripting or SQL injections, and access policies might not be properly implemented. An attacker can exploit these server-side vulnerabilities to inject malicious code snippets, called *infection vectors*, which, in turn, attack visitors through drive-by install or download attacks. Drive-by install and download attacks try to exploit client-side vulnerabilities to download or install malware, or lure the user into installing malware. If an attacker finds a server-side vulnerability that affects multiple web sites (possibly thousands), he can automate the exploitation, search for other vulnerable web sites, and launch a carefully crafted infection campaign in order to maximize the number of potential victims.

Recent reports by the security company Sophos [1, 2] show that in 2012 over 80% of all web sites attacking users were infected legitimate web sites, such as those of trade associations, nightclubs, television companies or elementary schools. All of these web sites had been altered, in one way or another, to attack visitors. In another case, in early 2013, web sites hosting documentation for software developers were modified to serve carefully crafted infection vectors that exploited client-side vulnerabilities, which were then leveraged as the first stepping stone in sophisticated attacks against Twitter [3], Facebook's engineering team [4], and Apple [5].

Major challenges in detecting infection vectors are that web sites become more and more dynamic and that their static content changes regularly, i.e., the underlying infection vector might not be clearly visible to analysis tools, or even to well-trained human security analysts. Adding new content to the web site, showing different, personalized advertisements, or even comments left by visitors are legitimate modifications. Yet, these modifications force the user to reevaluate the maliciousness of the web site, either through an automatic detection system or by manually inspecting any new content prior to its inclusion into the web site. Unwanted modifications to a web site on the other hand, i.e., changes from which a user might want himself to be protected, include defacements or the insertion of exploit code to infect visitors of the web site with malware. Previous work in the area of web evolution [6–10] suggests that web sites do not change randomly, but that they evolve constantly through small changes, and, if one takes into account personalized advertisements, a change might happen at every visit, which, in turn, makes it necessary to analyze the web sites on each visit.

The current state of the art in protecting a user from malicious content is mainly realized through blacklists, which are queried before the web site is rendered or retrieved by the web browser. The *Google Safe Browsing* list [11] is likely the most prominent example. By definition, blacklists are reactive, which is an undesirable property for any protection mechanism because a malicious web site can potentially stay undetected for an extended period of time. Once a web site is blacklisted, its operator must go to great lengths to remove his/her web site from the blacklist, although it might have become benign. Such a removal process can take a frustrating amount of time since it is often subject to some form of verification that the web site is now benign, a process that might not happen immediately. More important, however, is that each web site infected as part as an infection campaign needs to be identified and added to the blacklist even though the web sites attack visitors in the same, well-known way. Clearly, a proactive approach is preferable for both unknown and known infection vectors. On the other hand, scanning a web site proactively with online analyzer systems [12–14] is computationally very expensive, and would introduce delays up to multiple seconds per web site. Since such a delay is undesirable, it is unlikely that such a proactive approach would be deployed in a general setting, or that it would find its way into current browsers as a protection mechanism.

It is important to mention that, generally, the same infection vector is reused by an attacker and spread among a multitude of different web sites to maximize its impact; however, some parts of the infection vector might be randomized. Often, the infected web sites are from a single community, e.g., in a targeted attack on this community, they employ the same underlying software stack, or they share a web server that was attacked. Recent examples include attacks targeting installations of the Apache web server to replace the web server's executable with the backdoor "Linux/Cdorked.A" [15, 16], which injects code to redirect visitors of the web site to exploit pages. These compromised web sites are not necessarily targeted, they usually follow a simple pattern: the infection vector was inserted in the same or in a very similar way. Be it, as previously mentioned, through improper access control, exploited vulnerabilities in a web framework or application used by all web sites. Being able to identify an infection vector, instead of just detecting that the web site is malicious, can provide important feedback since the initial cause of the infection vector can be investigated much more easily due to additional information, such as commonalities in different observations of the same infection vector.

To overcome the limitations of current approaches mainly based on dynamic analysis of web sites, we introduce the Δ-system to identify malicious activity in a web site based on static analysis of the differences between the current and previous versions of the web site. We cluster these differences, determine if the introduced or removed elements are associated with malicious behavior, we identify the infection campaign it belongs to, pinpoint the actual infection vector, and automatically generate an identifying signature that can be leveraged for content-based protection.

The main contributions of this paper are the following:

- We introduce the Δ-system, which is based on a novel approach to statically analyze and detect web-based infection vectors, and which identifies infection campaigns based on features associated with modifications observed between two versions of a web site.
- We develop a tree difference algorithm that is resistant to tiny changes, such as typographical corrections or the small evolutionary modifications a web site undergoes.
- We develop a set of modification-motivated similarity measures to model the concepts of inserting and removing malicious behavior into and from a web site.
- We evaluate the Δ-system on a large scale dataset, containing 26 million unique pairs of web sites, to show its applicability in real-world scenarios in terms of infection campaign detection and identification capabilities.

2 Δ-SYSTEM DESIGN

The Δ-system, instead of trying to solve the problem of deciding if a web site is malicious or benign provides a solution to the search problem of finding new infection campaigns and identifying similar, known infection campaigns. Nonetheless we are still interested in deciding if a web site's current behavior is malicious or benign. Instead of analyzing web sites in their entirety, the Δ-system investigates only the difference between two versions of the same web site.

The main idea of the Δ-system is to identify if the change made to a web site altered the behavior of the web site, i.e., if we can be certain that the new version of the web site is malicious or benign, by investigating if the modifications are similar to already observed ones, such as modifications associated with an ongoing infection campaign. In order to identify the changes that were made to a web site, we need a base version, i.e., an older version of the same web site.

The analysis process of our system is described hereinafter, followed by a discussion on potential uses of our system, and the impact of deploying the Δ-system.

2.1 Analysis Process

The Δ-system's analysis process follows a simple four-step process, which is shown in Figure 1, and whose steps are:

1. Retrieval and normalization of the web site.
2. Similarity measurement with respect to a base version.
3. Cluster assignment of the similarity vector.
4. Generation of the identifying signature.

Evidently, a base version of a web site has to be available. In the case that a local base version does not exist, however, we might still be able to retrieve an older version through web archives, such as the Internet Archive[1] or a web cache provided by a search engine. This makes our approach applicable for web sites that are visited rarely and were no local base version is kept, if we accept the overhead to retrieve the base version from a remote archive. While this might seem counter-intuitive because of the potentially large time difference between the archived and current version, we show in our evaluation that this is indeed a possible alternative.

Following this brief overview, we discuss the important steps of the analysis process in more detail. First, normalization of a web site; second, how the similarity to the base version is measured; third, how the identifying signature is generated.

2.1.1 Retrieval and Normalization

First, we retrieve the current version of a web site, for instance the web site a user requested. Then, after we have retrieved the source code of that web site, excluding all external references, such as included scripts or frames, we perform multiple normalization steps: we normalize capitalization of all tags, we reorder attributes of each tag and discard invalid attributes, and we normalize the quotation of an attribute's value. We perform this normalization step to ensure that functional equivalent tags are treated equally during our evaluation, and that changes such as changing the capitalization of a tag or switching from single to double quotes do not affect our final results.

[1] Internet Archive, `http://www.archive.org`

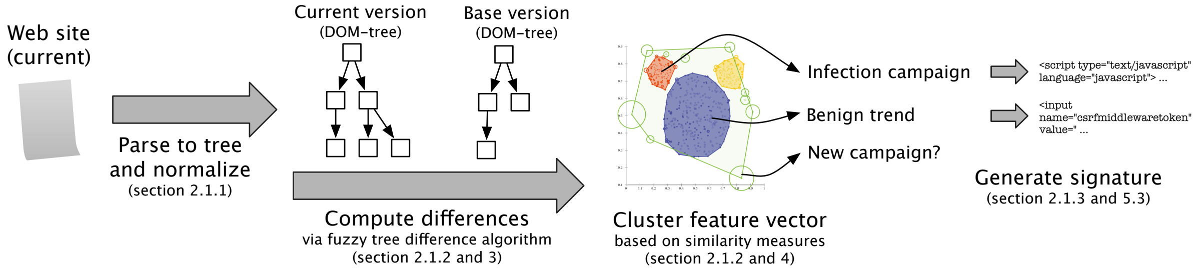

Figure 1: Analysis pipeline of the Δ-system.

2.1.2 Similarity Measurement and Clustering

Following these normalization steps, we measure the similarity to an already known (and normalized) base version. Measuring the similarity between two versions of the same web site in a meaningful way is non-trivial. The Δ-system first performs unordered tree-to-tree comparison via a novel algorithm that we introduce in Section 3. The algorithm extracts the nodes (or tags; subsequently, we use both terms interchangeably) from the Domain Object Model (DOM) tree of a web site that are different between base and current version. Second, based on the extracted nodes, we leverage a variety of different features to extract meaningful information from the two versions (described in detail in Section 4). The system then tries assigning these feature vectors to a cluster, or detects them as outliers, if they are not similar to any previously-observed modifications. Each different tag type, e.g., <input> or <script>, is treated separately, i.e., each type is assigned its own feature space; we do not project two tags of a different type into the same feature space. Additionally, due to the different nature of our features, where different distance metrics are essential for accurate cluster assignment, we perform consensus clustering for different groups: binary features, absolute and relative features are all treated as separate clustering instances. The cluster assignment and outlier detection process then distinguishes between three different cases:

- Assignment to an existing cluster:
  - Insertion or removal of an infection vector, if the cluster corresponds to a known infection campaign.
  - Legitimate modification, e.g., a version update of a library or the insertion of Facebook's like button, if the cluster does not correspond to an infection campaign.
- Detection as an outlier:
  - Potentially the start of a new infection campaign, if malicious behavior was inserted.
  - Potentially the end of a running infection campaign, if malicious behavior was removed.
  - A modification that is not of primary interest to us, such as a new article, template modifications, or a redesign of the web site.
- Formation of a new cluster (the similarity vector we are clustering and other vectors that are close, which were outliers before, put the number of total vectors in this area of the feature space above the threshold to form a new cluster, i.e., we observed the number of the same modification in the wild that we require to constitute a trend, c.f. Section 5.1):
  - New infection campaign, if the node was inserted and is associated with malicious behavior.
  - End of an infection campaign, if the node was removed and is associated with malicious behavior.
  - Legitimate modification, e.g., an update to a new, bleeding-edge version of a library or the content-management system used, such as Wordpress.

Upon cluster assignment of the similarity vector, we output the associated cluster, i.e., the corresponding trend (subsequently, we use these terms interchangeably). For instance, an infection campaign if the corresponding modification inserted or removed a known infection vector. Here, it is important to note that the detected clusters do not discriminate between removed and inserted nodes but treat them equally because we do not leverage the notion of removal or insertion in the feature computation, but attach it to the vector as "external" meta information that is not used during clustering. This supports the detection of removal and insertion of the same trend with the exact same cluster in both cases and, therefore, increases robustness of our system.

The Δ-system does not provide detection capabilities for malicious behavior on its own, but rather relies on an external detection system. This detection system is queried once a new cluster is formed to identify if this observed trend constitutes a malicious or a benign cluster. In order to guarantee a high likelihood, we "bootstrap" each cluster by querying for 10 random samples, and acquire a consensus decision for the returned labels. For instance, a new cluster is observed and 9 out of 10 of the random samples from this cluster have been assigned the label malicious, then the Δ-system will assign the label malicious to any new observation in this cluster.

2.1.3 Signature Generation

For each of the identified trends, we generate a signature that matches the textual representation of all of the nodes assigned to a cluster (e.g., a cluster describing "<script src='http://$random-url/exploit.js'>"). This signature is generated by simply interpreting the textual representation of each node as a deterministic finite automaton, merging them together, and calculating the minimal version, which can be done in polynomial time. The resulting DFA can then be translated into a regular expression that can be used by intrusion detection/prevention systems[2].

Such an identifying signature is, generally, an under-approximation of the actual (unknown) signature. For instance, in the above example the URL is randomized. Here, the generated signature only describes the observed samples, i.e., where $random-url might be "a.com" or "b.org", while the trend could be more general and also include "c.net". Leveraging only the identifying signature would miss web sites that follow the same trend, i.e., web sites who might serve the same infection vector. While this is of no concern in the case of

[2] Although generated signatures match normalized tags by default, it is trivial to normalize incoming data in the same way and match arbitrary tags that follow the same trend.

leveraging the Δ-system for every request (here, we would assign a similar, but unobserved, tag to the same cluster), it can be an issue if the generated signature is used as input to other tools. A possible remedy is to generalize the signature and to introduce a widening operator to describe the different parts of the nodes following this trend. For instance, one could simply widen 5 different characters at the same position in 5 different random samples picked from a cluster to an over-approximating wildcard. An over-approximation, however, is also likely to introduce incorrect matches, which is why we recommend using the Δ-system if no exact signature matched.

2.2 Use Cases

We see the Δ-system to be deployed in two main scenarios: next to a web crawler to actively search for new infection campaigns, and next to a proxy to identify infection campaigns (passive) or to improve user-protection (active). Additionally, there is a third, minor scenario: providing feedback on evasions of detection systems. Subsequently, we describe all three use cases in more detail, however, in the remaining of the paper, we focus on the first use case: paired with a web crawler.

The most interesting use case, in our opinion, is the active identification of new infection campaigns. In this case, one deploys the system side-by-side to a web crawler. While the web crawler retrieves potentially interesting web sites multiple times over a given period of time, our system analyzes the differences. When our system detects a new cluster, i.e., a significant number of very similar modifications, an external detection system then decides if this change is associated with malicious behavior or not. If malicious behavior was introduced then we found a new infection campaign and we can generate the identifying signature for this cluster. Based on the elements of the cluster, we can then pinpoint the infection vector (e.g., identify parts of the tag that are common among all web sites in that cluster) and investigate other similarities manually (e.g., only online stores running a specific version of the PHP application osCommerce were infected). Starting from those similarities, it is then possible to: generate a more precise fingerprint for the campaign, find other infections via search engines, and estimate the scope of an infection campaign.

The second envisioned deployment of the Δ-system is the extension of a web browser or a proxy. In most cases, the browser or proxy already caches visited web sites for performance reasons. Moreover, in security-sensitive environments, it is very likely that a detection system (e.g., an anti-malware engine) is already in place to ensure that only benign web sites can be accessed by the user. Such a detection system can be leveraged by the Δ-system to analyze inserted tags. The system can complement these tools to prevent repetitive scanning of web sites, to improve user experience by increasing analysis performance, and to provide insight into targeted attacks. For example, small changes a user might encounter include automatic page impressions counter, updated weather or date information, or the output of the processing time to render the web site on the server's side. While previous work requires the reevaluation of the entire web site, the Δ-system can identify these changes as benign much more easily. It is even possible to obtain more accurate results with our system than with the detection system, e.g., if it is based upon simple detection methods, such as fingerprinting of known malicious scripts, or if the detection system is being evaded. Additionally, once a malicious web site is identified, the Δ-system can verify that a malicious modification was removed and that the web site is now benign. Particularly, if an infection campaign is dormant or the exploit page is offline, dynamic analysis systems detect that the web site is benign because it does not detect any malicious behavior. Since the Δ-system is purely static and verifies that the malicious content was removed, it does not have this disadvantage.

Lastly, the Δ-system can also be leveraged to detect evasions and bugs in detection systems and online analyzers. For example, if the analyzer is dynamic, but behaves differently than a standard browser in even a single case, then malware can fingerprint the detection system. Such a fingerprinting method allows the attacker to thwart detection much more easily, for instance, without having to utilize a blacklist of the IP addresses used by the online analyzer. Leveraging our system, we can detect these evasions when they are introduced. The system can pinpoint the changed content precisely and, by doing this, support the developer in identifying the reason why the analyzer is behaving differently, correcting the corresponding bug, and preventing further evasions leveraging the same bug.

3 FUZZY TREE DIFFERENCE

First, to be able to measure the similarity between two web sites in a meaningful way, we need to define the notion of difference. We are primarily concerned if a web site behaves in a benign or malicious way. To this end, we need to understand what modifications to the content can result in behavioral changes, and how we can isolate the modifications from other parts of the web site that have no effect on the overall behavior. We identify these interesting parts by leveraging the hierarchical structure of a web site and interpreting a web site through its DOM tree.

Previous work introduced various algorithms to detect the semantic change in hierarchical structured data. The main idea behind HTML, i.e., describing how to display data instead of describing the semantics of the data itself, renders nearly all introduced XML-centered approaches unsuitable to extract meaningful information about the modifications. An often made assumption is that the underlying tree structure has a significant semantic relationship with the content, which is not necessarily the case for HTML. Moreover, leveraging standard maximum cardinality matching on cryptographic hashes and simple edge weights of 1 (based in the nature of cryptographic hash functions), any change would be visible, including very small changes that are uninteresting to us, such as single character or word changes and legitimate evolutions. We denote such a tree-to-tree comparison as *not tiny change resistant* or *not fuzzy*. To solve this problem, and to identify interesting modifications made to a web site more precisely and more efficiently, we generalize the previous notion of tree difference algorithms and introduce a similarity weight. We refer to our algorithm as the *fuzzy tree difference algorithm*, which is heavily influenced by the unordered tree-to-tree comparison algorithms by Chawathe et al. [17] and Wang et al. [18]. Such a fuzzy algorithm is necessary when comparing web sites that have evolved over an extended period of time, e.g., have been edited constantly over a two week period. Otherwise, the sheer number of remaining nodes to analyze makes it infeasible to leverage computationally expensive features with reasonable performance overhead.

While we provide a formal description of the algorithm in Algorithm 1, we give a brief informal description first: the algorithm expects three parameters, T_1, T_2 and t_r. T_1 and T_2 are normalized DOM trees, i.e., all tags are capitalized in the same way, all attributes occur in the same order and their values are enclosed in the same way (quote-wise). t_r is the threshold value for the similarity measurement, and can range from 0 to 1. Starting from the trees T_1 and T_2, we create a temporary graph to match pairs of similar nodes through maximum weighted bipartite graph matching (Hungarian algorithm [19]). This graph is constructed by inserting every node of T_1, then inserting every node of T_2. For each node from T_2, we connect it with an edge to every node from T_1 that has a similar fuzzy hash value (i.e., the Jaro distance of both hashes must be greater or equal to t_r) and that takes the exact same path (in the sense of unordered tree-traversal) as the node from T_2. The edge's weight is equal to the similarity measured through the Jaro distance between both hashes (i.e., at least t_r). Additionally, we color all matched nodes blue. In the last step, we remove the corresponding matched nodes from the trees T_1 and T_2 and output a list of removed (remaining in T_1) and inserted (remaining in T_2) nodes.

While the reason for coloring nodes might not be obvious, later on, we leverage the color of a node in the remaining nodes of T_1 and T_2 in our similarity measures to detect a matching asymmetry, i.e., if a tag with a very similar hash and the same path from the root node was matched, such as a template that was used more often in T_2 than in T_1.

The implementation of the Δ-system under evaluation leverages *ssdeep* [20] as the fuzzy hash function and a threshold of 0.99 for the Jaro distance [21] (which is normalized to 0 to 1, i.e., we require very similar tags). Similar to cryptographic hash functions like MD5 or SHA, a fuzzy hash function, such as *ssdeep*, maps arbitrary long values to a short hash. In contrast to cryptographic hash function, however, a fuzzy hash function maps similar values to similar hashes that can be then used to measure their similarity. This property allows us to efficiently compare nodes of the DOM tree or their content regardless of their actual length, which otherwise might be computational too expensive when using standard string similarity measures for longer tags or content. We selected the Jaro distance function to compare two hash values because it is a simple string similarity measure originally introduced for duplicate detection by Jaro [21] and best suited for short strings while accounting for exactly matched characters as well as transpositions, therefore it quantifies the similarity of fuzzy hashes for similar data accurately.

In general, a threshold value of 1 when used with a cryptographic hash function is equivalent to standard unordered tree-to-tree algorithms. On the other hand, a threshold value of 0 regardless of the hash function is equivalent to comparing every element to every other element and impractical for any modern web site due to the sheer number of possible combinations, which is why a reduction of potential matches is essential.

3.1 Example

An example of the tree difference algorithm is shown in Figure 2. The source code of a simple base version and current version of a web site are shown in Listing 1 and Listing 2 respectively. Two modifications to the source code were made: first, a head tag including a script tag with an external source URL was inserted, and, second, a typographical mistake in the class of the p tag was fixed and one word in its content was changed: "foo" was replaced by "bar". Figure 2 illustrates that for a standard tree difference algorithm the modified p tag would, correctly, constitute a modified p tag (the removed p tag is marked with a red chessboard pattern, while the inserted p tag is marked with a green diagonal pattern). However, since we are interested in severe changes and modifications associated with behavioral changes, these tiny changes are uninteresting to us, and, like the example shows, they are discarded by our algorithm.

Algorithm 1 Fuzzy Tree Difference

```
function FuzzyTreeDifference($T_1, T_2, t_r$)
    $G \leftarrow$ Graph
    for all $n \in T_1$.nodes do
        $G \leftarrow G$.insert_node($n$)
    for all $n \in T_2$.nodes do
        for all $m \in T_1$.nodes do
            if path($m$) = path($n$) then
                $d_{(m,n)} \leftarrow$ jaro(hash($m$), hash($n$))
                if $d_{(m,n)} \geq t_r$ then
                    $G$.insert_node($n$)
                    $m$.color $\leftarrow$ blue
                    $n$.color $\leftarrow$ blue
                    $G$.insert_edge($m, n, d_{(m,n)}$)
    $M \leftarrow$ max_weight_matching($G$)
    for all $(m, n) \in M$ do
        $T_1$.remove_node($m$)
        $T_2$.remove_node($n$)
    return $T_1, T_2$
```

4 SIMILARITY MEASURES / FEATURES

The most interesting part of web sites from a malicious code point of view is described by the HTML markup language: JavaScript, inline frames, or the use of plugins. Most research on document similarity, however, assumes that markup language is not of major interest and that it can be removed without substantial loss of information. For detecting infection vectors, this assumption does not hold. Essentially, this violation makes applying existing work in document similarity for identifying infection vectors impractical, because core elements are discarded.

Therefore, we introduce our own similarity measures. Once we have extracted the different tags between two versions of a web site, we can map each tag into the feature space in which we cluster similar changes together. In this section, we describe the features we are using and the intuition behind them. Each of our features we apply on multiple levels (where applicable): the whole tag and for every value of its attributes.

4.1 Template Propagation

First, we introduce the template propagation measure, a binary feature that simply models what content was introduced or removed from the web site in terms of their similarity to previous DOM tree nodes, i.e., it captures the concept of reused templates by checking if a node exist already in the base version, but are unmatched, e.g., because there are more matching candidates than actual matches are possible. Since all matched nodes in the output T_1 and T_2 of our tree difference algorithm are colored blue, we can simply set the value of this feature to 1 if the node is blue and 0 if it is not.

The motivation for this measure is that many web sites, for example blogs, use templates when publishing a new article or when showing a new comment, classified, or advertisement. Detecting that a template is repeated allows us to model the degree to which a web site has drifted away from expected changes, e.g., in terms of character count distributions for a blog with articles written in English.

4.2 Shannon Entropy

Second, we leverage the Shannon entropy as a feature of information in a tag or an attribute's value. Two different features are derived from the Shannon entropy: (a) the absolute Shannon entropy, which is dependent on the length of the string, (b) the normalized Shannon entropy, i.e., the absolute Shannon

```
<html>
  <body>
    <p class="sumamry">
      [...] foo [...]
    </p>
  </body>
</html>
```

Listing 1: Base version, source code.

```
<html>
  <head><script
    src="http://url/malicious.js">
  </script></head>
  <body><p class="summary">
      [...] bar [...]
  </p></body>
</html>
```

Listing 2: Current version, source code.

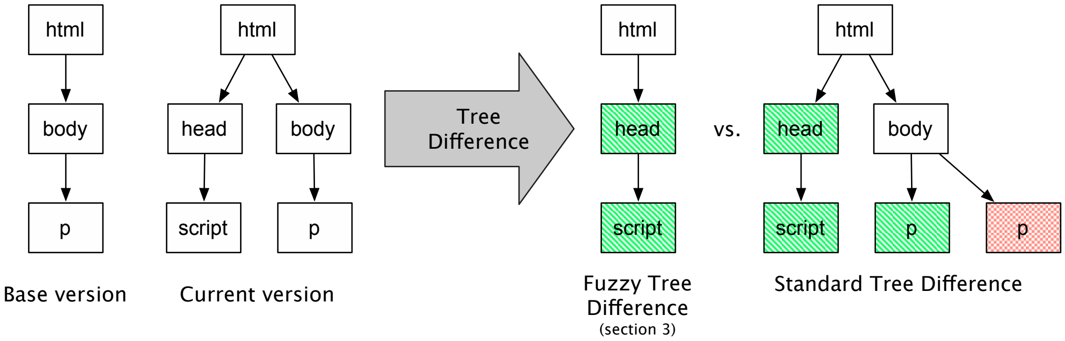

Figure 2: Comparison between a general tree difference algorithm and the fuzzy tree difference algorithm (section 3). Nodes with a green diagonal stripes pattern denote nodes that were detected as inserted, while nodes with a red chessboard pattern were detected as being removed.

entropy divided by the ideal Shannon entropy of a string of the same length (i.e., it is normalized to the interval from 0 to 1).

Our intuition behind the Shannon entropy is to measure the distance on how far the tag or attribute is away from a random source. For instance, to model that the URL in a "src" attribute of a <script> tag was generated by a random source.

4.3 Character Count

In the third set of features we employ a character count. The first feature in this set simply measures how often a single character occurs in the tag or the attribute's value and discriminates between upper- and lower-case characters. The second feature also measures how often a single character occurs, however, it ignores capitalization and counts an "A" as an "a". The third feature follows in simplicity and is the count of each digit. A fourth, fifth and sixth feature are taking advantage of the same method, but are performed on the fuzzy hash value (ssdeep in our implementation) of the tag or attribute instead.

Beyond these six features, we are also computing relative features for both of those two sets, as we did already in case of the Shannon entropy. Alike to the Shannon entropy features, the motivation behind these features is to model the character and digit distribution in a string.

4.4 Kolmogorov Complexity

The third set of measures we introduce is based on an approximation of the upper-bound on the Kolmogorov complexity [22]. Kolmogorov complexity denotes a complexity measure specifying the lower-bound of text necessary to describe another piece of text in an algorithmic way. One of the most important properties of the complexity is its incomputability. An upper-bound on the other hand is easy to compute by taking the length of the text compressed by any lossless compression algorithm. Since these features are based on a second information theoretical measure, next to the Shannon entropy, it is necessary to emphasize that they are complementary in our scenario: on the one hand, Kolmogorov complexity is conceptually different from the Shannon entropy, and on the other hand, we approximate the Kolmogorov complexity up to (at best) an additive constant.

These measures exploit the upper-bound and the fact that compressing already packed data results in nearly no benefit, in order to measure a change that introduces packed or encrypted data, such as malicious data trying to evade detection.

We introduce, again, two different features based on computing an upper-bound of the Kolmogorov complexity. First, the absolute upper-bound on the tag extracted by our tree difference algorithm; second, the ratio of the upper-bound over the length of the string. In case of very short strings, the upper-bound might even take up more space than the actual string.

4.5 Script Inclusion

Scripts included in web sites are the most prominent way to infect a user with malware, but they are also used legitimately. Differences exist between how malicious scripts and legitimate ones are used and included. For instance, malicious scripts are rarely including local files; instead, they usually include from an external source or provide the source code directly. The following two binary features model these differences.

4.5.1 Absolute Source URL

The enduring rise of content-delivery networks, which are often heavily relying on load-balancing based on the domain name system (DNS), lead to scripts being included much more often with an absolute and external source address in legitimate cases than it was the case prior to the predominance of these networks (due to potential compatibility issues if scripts are included differently). It is important to understand if a web site is hosted on a content-delivery network since it bears the reasoning that these web sites are generally much more optimized than personal web sites, to save on bandwidth on account of the smaller size. This then has an impact on the importance of other features. This feature is also binary; it is 1 for an absolute non-external source URL and 0 otherwise.

The many legitimate use cases of including scripts from an absolute URL suggest that this feature will not have a discriminatory impact on its own; rather, it supports other features by modeling the inclusion-style in a web site. The notion of a single inclusion-style roots in previous work by Nikiforakis et al. [23], which suggests that web sites follow the same inclusion patterns, i.e., the distribution of how scripts are included is biased toward either relative or absolute inclusions, and only rarely uniform.

4.5.2 External Source URL

While the last feature is a bias function to judge the use of scripts with an absolute source, the next feature is a bias function for the concept of external source URLs. It is important to mention that, if an external script is included, assuming the external domain is maintained by a third party, then the web site operator has to trust that the third party providing the external script will not insert any malicious code.

4.6 Inline Frames

Similar to the features to model the use of script tags, the following three binary features try to model the inclusion of malicious inline frames, by looking into properties that are uncommon for benign inclusions.

4.6.1 Absolute and External Source URL

Likewise to the nature of the source URL features for scripts, these features give an intuition on the use of inline frames. Both features are identical to their script sibling, but they examine <iframe> tags instead of <script> tags.

Their motivation follows closely the motivation for the script features: i.e., the feature for absolute source URLs is supporting other inline frame measures as a bias function. In the past, inline frames with an external source address were often used to include either advertisements or third party widgets. Recently, those moved to inline JavaScript or embedding plugins directly. Adversaries, on the other hand, still use these frames because they allow for easier fingerprinting and support delivering different infection vectors per user, for example depending on the browser's patch-level, installed plugins, or by obfuscating each reply differently. This fine-grained control helps the adversary to maximize the attack efficacy while reducing the likelihood of detection.

4.6.2 Hidden Frame

Beyond absolute and external frames, another indicator for malicious content being included exists: hidden frames. Legitimate frames are generally made visible to the user. Adversaries on the other hand prefer that the included web site is invisible, which is why they often resort to setting width and height of the frame to a low value, so that the frame is visually hard to spot for a human. We investigated a random sample of 10,000 inline frame tags that we extracted from our dataset and found that legitimate inline frames are often set to a width and height of larger than 100 and rarely hidden (the style attribute "display: none" is rarely used). We model this phenomenon, assuming that a majority of malicious inline frames uses a much smaller area of screen space, by restricting width and height of our feature to a maximum of 15 pixels. The feature is simply 1 for hidden frames and 0 otherwise.

5 EVALUATION

Generally, the problem we are trying to solve is an instance of *knowledge discovery in databases* [24, 25]. More precisely, when searching actively for infection campaigns next to a web crawler, we are interested in detecting outliers, i.e., novel changes, and the appearance of clusters, i.e., when a new trend is observed, for instance an infection campaign. While various clustering algorithms can be employed, the design of our system encourages the use of an algorithm that detects local outliers. Additionally, the distribution of a cluster can differ from the distribution of any other cluster, particularly for clusters with a low member count, i.e., it is not reasonable to assume that all changes follow a very similar distribution in the feature space. Since we are primarily interested in the formation of new clusters, when it is even less likely that this assumption will hold, centroid- or distribution-based clustering algorithms such as k-means (which will give spherically shaped clusters) or expectation-maximization (e.g., Gaussian mixture models) are unlikely to provide any valuable insight on new infection campaigns early enough. To counter this issue, we adopt a variant of the density-based clustering algorithm OPTICS (Ordering Points To Identify the Clustering Structure, by Ankerst et al. [26]), namely OPTICS-OF (OPTICS with Outlier Factors) by Breunig et al. [27].

5.1 OPTICS-OF

The OPTICS-OF algorithm takes two parameters: the maximum distance for a cluster and the minimal number of vectors necessary to form a cluster. In the scenario of trend analysis, the maximum distance corresponds to the similarity of a change, while the minimal number of vectors necessary describes the number of instances of a change we want to observe before we consider it a trend, i.e., before we want to verify that we found a previously-unknown infection campaign.

The algorithm works, in essence, as follows: if two vectors in the feature space are closer than the maximum distance, then they are directly density-reachable. If at least the minimal number of vectors are directly density-reachable from a vector, then this vector is a core object and forms a cluster. A cluster does not only contain directly density-reachable vectors from this core object, but is defined transitively, i.e., it contains all vectors that are directly and transitively density-reachable from the core objects. Therefore, an outlier is either not density-reachable to any other vector at all, or only density-reachable to a number of vectors, where none of the vectors is a core object, i.e., none of the vectors forms a cluster. In our experiments, we require 10 similarity vectors that are directly density-reachable to form a cluster.

5.2 Dataset

First, in this section, we describe in detail what constraints we imposed on our dataset, why these constraints were imposed, and how we obtained our dataset. In general, it is a challenging problem to obtain a representative sample of different and distinct malicious web sites. Invernizzi et al. [28] have shown that this is even the case when only mediocre toxicity[3] is required, i.e., it is even difficult for a dataset with a small but non-negligible percentage of malicious web sites. This poses a problem for collecting our dataset because we desire moderate toxicity and diversity among malicious infection vectors to verify that we can correctly, and without bias, identify similar trends, and by this, similar infection campaigns. Moreover, to discard trivial cases, the requirements on the web sites in our dataset are even more restrictive:

- Web sites must have been set-up for a legitimate reason, i.e., we are interested in landing pages and not interested

[3] Toxicity measures the maliciousness of dataset and simply corresponds to the fraction of malicious samples in a dataset over all the samples of the dataset.

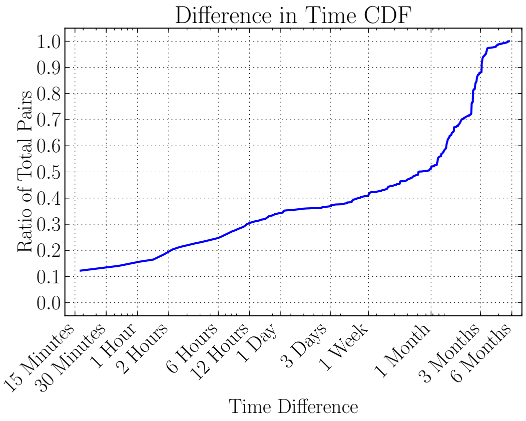

Figure 3: Overview of the difference in time between base and current version of a pair in the final dataset.

in exploit pages. Exploit pages denote web sites that are set up by an adversary to exclusively deliver malicious code, while landing pages denote the infected legitimate page. We enforce this restriction because recent work establishes that legitimate web sites are nowadays the primary target and because other approaches by Provos et al. [11, 14], Ratanaworabhan et al. [29], Curtsinger et al. [30] or Seifert et al. [31] are already able to detect purely malicious web sites with outstanding accuracy.

- Two distinct versions of a web site are required, i.e., a web sites must have been modified (legitimately or maliciously) to constitute a realistic sample.

We obtained our dataset by crawling the web from January 2013 to May 2013 via a 10-node cluster of custom crawlers running an adaptive fetch schedule with a recrawl delay of at least 15 minutes and an exponential back-off delay (multiplied by a constant factor of 10 if no change was observed in a recrawl and with a strict maximum of one week). The hourly seed of URLs for our crawler contained web sites that were already present in our dataset and also Yandex's search engines results for Twitter's trending topics. Additionally, to counter the problem of low toxicity and prevent a bias toward benign web sites, we injected a total of 2,979,942 URLs of web sites into our crawl seed that the Wepawet online analyzer [12] had analyzed previously, starting with samples observed in the beginning of January 2013 and ending with samples observed at the end of April 2013. In total, after removing exact duplicates and restricting the number of pairs per unique URL to a maximum of 10, our dataset spans a size of 700GiB and 26,459,103 distinct web site pairs from 12,464,920 unique URLs. A distinct pair denotes a pair where both versions are different from each other in respect to the SHA256 checksum of their normalized DOM tree.

The time difference, before a change between base and current version of a web site was observed, is shown in Figure 3. The average time difference of our pairs is 4 weeks, with 80% of all pairs being 2 hours or more apart, 70% being 12 hours or more apart, 60% being 7 days or more apart, and 50% being 20 days or more apart (median). Since not all web sites have been recrawled after exactly 15 minutes, we can only observe that a change happens in at least 16% of the web sites in a 15 minute interval after a visit.

5.3 Case Studies: Identified Trends/Clusters

In our experiments, we identified a total of 67,038 different clusters, with the majority of clusters having 30 or less elements.

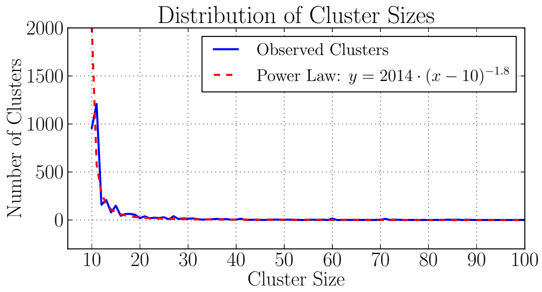

Figure 4: Overview of the number of different elements that are in each cluster.

Figure 4 depicts the final distribution of cluster sizes we observed in our experiments. Evidently, the observed distribution follows closely the power law function: $y = 2014 \cdot (x - 10)^{-1.8}$. In addition, we observe that the total sum over all cluster sizes is less than the number of distinct tags that we have analyzed. This is the case because any remaining tags are still considered to be outliers and do not constitute a trend yet. As a matter of fact, both the close resemblance to a power law function and a non-negligible amount of outliers are expected, because some changes are only made to a limited number of web sites, e.g., very similar articles might be posted to less web sites than we require as a lower-bound to constitute a trend, and also because our view of changes is limited by the seed and link expansion of the web crawler, i.e., it is possible that we only observed a subset of the true instances of each unique trend.

We feel that it is important to understand what a single cluster is actually describing, and we provide different examples about what tags have been clustered together. Therefore, we investigate two clusters in more detail. Although both clusters are low-count clusters, i.e., relatively small, their small size actually illustrates that the Δ-system does detect when a trend reaches a significant distribution and that it does not rely on an unreasonable large number of observations of a single trend.

The first example we discuss is an actual infection campaign that we have observed in the wild, an instance of a redirection to a Cool Exploit Kit installation. In contrast, the second example we discuss corresponds to a cluster describing the change in cross-site request forgery tokens.

We selected these two clusters manually by filtering clusters based on the generated signature with simple heuristics that suggest malicious behavior, such as external scripts that are included with a random component or JavaScript with a non-negligible ratio of digits over characters (suggesting obfuscation). Clearly, these and other heuristics can also be leveraged to order clusters according to "levels of interest" or to remove clusters that are likely uninteresting and should not be analyzed manually by an analyst.

Other trends we observed, but will not discuss in detail, include the modification of Facebook Like buttons (the backlink URL changes), a version update for the JQuery library served for blogs hosted on Wordpress.com, or the insertion of user-tracking tokens.

5.3.1 Cool Exploit Kit Infections of Discuz!X

One of the most interesting clusters, which shows the applicability of the Δ-system in practice, describes an infection vector used to redirect to a specific infection campaign that uses the Cool Exploit Kit to distribute malware. This in-the-wild infection campaign was found at the beginning of April 2013 in a set of 15 different web sites from the following 10 unique URLs:

- http://att.kafan.cn

- http://frozen-fs.net
- http://jses40813.ibbt.tw
- http://ppin888.com
- http://www.dv3.com.cn
- http://www.kxxwg.com
- http://www.ruadapalma.com
- http://www.sdchina.cn
- http://www.wlanwifi.net
- http://www.yysyuan.com

Once we verified that the cluster was indeed malicious, we investigated the underlying commonalities between them. We found that all web sites were using the discussion platform "Discuz!X" [32]. Discuz!X is an Internet forum software written in PHP and, according to the Chinese National Radio [33], the most popular Internet forum software used in China. Clearly, these infections are part of the same infection campaign. Additionally, such a strong common ground suggests that the infection is likely to be rooted in a security vulnerability in the Discuz!X software, and it provides support identifying the cause and a removal method.

Listing 3 shows the respective generated signature of the infection. For this infection campaign, we did not observe any differences in the tags that were clustered together.

```
<script type
    ="text/javascript" language="javascript">
  p=parseInt;
  ss=(123) ? String.fromCharCode : 0;
  asgq=" [4036 character obfuscated string] "
      .replace(/@/g,"9").split("!");
  try { document.body&=0.1 } catch(gdsgsdg) {
    zz=3; dbshre=79;
    if(dbshre) { vfvwe=0;
      try { document; }
      catch(agdsg) { vfvwe=1; }
      if(!vfvwe) { e=eval; }
      s="";
      if(zz) for(i=0;i-1374!=0;i++) {
        if(window.document)
          s+=ss(p(asgq[i],16)); }
      if(window.document) e(s); }}</script>
```

Listing 3: Cool Exploit Kit infection vector.

Beyond the inclusions of infection vectors pointing to an installation of the Cool Exploit Kit observed in all pairs, one web site (http://frozen-fs.net) also included an infection vector that tried to infect visitors via an installation of the Blackhole exploit kit.

The domain that included the Cool Exploit Kit and the Blackhole exploit kit, "frozen-fs.net", was not cleaned up, and we observed that it was suspended by the provider 27 days after we detected the infection.

5.3.2 *Cross-Site Request Forgery Tokens*

A second interesting low-count cluster we found during our evaluation models variations in cross-site request forgery tokens in deployments of the Django web application framework. In total, we identified a similar modification among 17 different pairs of web sites. Each web site used form-based cross-site request forgery tokens and used the same identifier for a hidden form field, namely "csrfmiddlewaretoken". For every pair, the attribute features did not diverge for the name attribute, while all were different for the value attribute. Nonetheless, the Δ-system clustered them together, since the random entropy was nearly constant for the value attribute among all observed removed and inserted instances. The entropy was nearly constant for the normalized case as well as for the absolute entropy features. The exact identifying signature for that cluster is shown in Listing 4.

```
<input
 name="csrfmiddlewaretoken"
 value="(JhD3IwCXcnnpRtvE42MN6r8dOBOWRoxG
        |hH4f6eOMCOTEYFORYoXFRDaTLzym61O2
        [...]
        |DNczoWjeN1nK6nq3whXYpSSnZGdxxOOg
        |F9yLSOjNUXIURsXDRqxS5NVW7qXfWsgf)"/>
```

Listing 4: Cross-site request forgery token; | denotes an or.

We feel that this observed trend constitutes a perfect example in which the limitations of the signature generation stick out and where the Δ-system shows its robustness by clustering these changes correctly together. While the signature can detect all observed instances correctly, it is clear that when trying to match new versions of a web site with the signature we would fail to identify the changed token value correctly since the value will change to a new, unobserved random value.

5.4 Performance

In order to judge the actual applicability of our system in practice, a performance analysis is necessary. We show in Figure 5 that the performance of the Δ-system allows for deployments in real-world scenarios. However, corner cases exist that could impact an actual deployment, if the difference between the base and current version of a web site is particularly large. We performed a manual in-depth analysis of the system that highlighted the actual performance bottleneck of our system: close to 80% of the time when analyzing the two versions was spent by the Python library BeautifulSoup to parse the HTML structure of a page. Although, the number of changes made to a web site plays the most important role in analyzing the changes, pairs that took longer than 3 seconds to analyze were exclusively web sites that sent data in an encoding different than specified. BeautifulSoup tries to take care of this and follows a code path that can get multiple thousand function calls deep, and easily hits the recursion limit of CPython. While we increased this limit in our evaluation to keep these pairs and prevent a dataset bias, the particular functions are actually tail-recursive and, therefore, can be expressed iteratively (thus, removing the necessity of allocating stackframes). However, the abstruseness of the involved functions prevented us from doing the very same in a reasonable amount of time. Regardless of these (still outstanding) engineering challenges for a general deployment, we could analyze a single pair in a median time of 0.340 seconds and in an average time of in 2.232 seconds. It is also evident that we finished each analysis in at most 20 seconds, regardless of the aforementioned problems encountered in BeautifulSoup.

These results, when taking into account that 60% of our data is 7 days or more apart (c.f. Figure 5), support our claim that the Δ-system does not necessarily need to keep a base version locally, but could rely on public archives like the Internet Archive or a web cache by a search engine. Nonetheless, we strongly recommend keeping a local version to prevent an additional delay in fetching the web site and to prevent running into the problem of a potentially outdated or even non-existing version on the side of the public archive.

6 LIMITATIONS

Similar to other static analysis approaches leveraging machine learning, our approach has some limitations, which can be used to evade detection. This section discusses these limitations and how they could be managed in a real-world deployment of the Δ-system. First, we introduce a limitation called

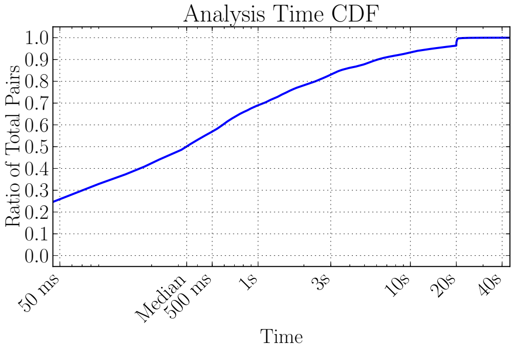

Figure 5: Overview of the ratio of web site pairs that have been completely analyzed in our experiments in less than x seconds.

Step-by-step Injection. Second, we will briefly discuss the *Evolution of Infection Vectors* as a major fundamental problem in detecting malicious code. Lastly, we discuss the trade-off between dynamic and static analysis and the limitations of either approach.

6.1 Step-by-step Injection

It is possible to circumvent the Δ-system by adding malicious code in small steps, i.e., in a series of modifications where each step on its own gets detected as being benign, while the aggregation is malicious. For example, an attacker could build an infection vector that delivers these small steps depending on a cookie or the visitor's IP address to keep track of the client's previous version. However, we argue that the scope of such an attack is heavily limited because: (a) it requires an attacker to be able to inject code that is executed on the server-side, otherwise detection is possible because it has to be done client-side, and the attack can also be impeded or even avoided by keeping the first version instead of updating the stored base version with every visit; or (b) the DOM tree will be modified online, for example through JavaScript. In the first case, an important and drastically scope-reducing factor is that the vulnerability needs to support such an iterative process, where, e.g., memory regions are shared among browser tabs or between multiple visits to the same web site, which is highly unlikely given the strict separation current browser sandboxes enforce. In the latter case, on the other hand, we suggest to analyze the web site on every mutation event[4], i.e., by considering the web site that was modified online as a new current version and comparing it to the stored base version.

6.2 Evolution of Infection Vectors

Detecting malicious code is an arms race and malicious web sites are no exception. Malware developers are trying to evade detection systems to gain the upper hand, while detection system developers are trying to catch and prevent these evasions. Previous work on evasions motivated the search for better and different detection systems [34]. More advanced obfuscation [35], encryption, poly- and metamorphic code [36] and virtualized environments have become more common in response to these improvements and impeded detection systems. With more approaches being able to handle these cases, it is to be expected that malware and infection vectors will evolve and successfully circumvent available detection systems. While retraining the machine learning algorithm on a more recent dataset is often a possible approach to counter the evasion problem, it is only a near-sighted solution to counter the dataset shift, as malware will deviate more severely, up to the point where the features will not model the underlying problem anymore. Even in cases where the features are not publicly known to an adversary, it is possible to partially derive these by probing the system carefully, which then will either allow for successful evasion of the system or (on re-training) increase the false positive and false negative rate because of misclassification due to minuscule differences between legitimate and malicious code in the feature space. Both cases are obviously not desired for a detection system, however, it is a general problem of all approaches employing machine learning [10,11,13,14,31,37–39], and it is generally only countered reliably by adapting to a new feature space, which we leave for future work.

6.3 Dynamic vs. Static Analysis

The Δ-system in its current form is a purely static analysis system, while, at the same time, the Internet is becoming more and more dynamic. While one might think that static analysis is inferior to dynamic analysis here, this is not the case. Instead, our system complements dynamic analysis systems: it detects trends/infections statically and can forward the interesting trends/infections to dynamic analysis systems that extract further information.

Our motivation to rely on a purely static analysis is based on multiple reasons. First, dynamic analysis is not necessarily useful at the early stage in which our system operates, i.e., trends that change the behavior and are interesting to us show themselves first with static content changes, rendering dynamic analysis (currently) unnecessary. A second argument against dynamic analysis for the Δ-system is that it, for instance by instrumenting embedded or included JavaScript to modify the DOM tree to retrieve a "final" version of the web site, under-approximates the behavior of the web site to this specific execution environment and might yield a potentially incomplete or untrue representation of the DOM tree. It also poses the questions of when to consider the DOM tree "final", i.e., when to take a snapshot. Consequently, it might then be possible to evade the trend detection step in the first place. Additionally, we might also miss infection campaigns that are statically present in the web site, but are removed dynamically or are inactive (for us). For example, servers could be unavailable (for us) or code might not be loaded (for us), we could be fingerprinted, the IP address of our analysis system might be in a region of the world that is not affected or simply because the user-agent of our browser does not match a (unknown) regular expression. The third argument in favor of static analysis is that it can be considerably faster than dynamic analysis, which, in turn, allows us to leverage more computationally-expensive features to increase trend detection accuracy.

Lastly, while the trend detection step is purely static, to detect malicious behavior, the Δ-system relies on an external analysis system that might very well use dynamic analysis. Generally, we do not impose any limitations on this detection engine but that it can detect malicious behavior.

7 RELATED WORK

In the following we discuss related work in areas tangent to our research, such as web dynamics and the detection of malicious code. To the best of our knowledge, no prior work exists that actively searches and finds previously unknown infection campaigns.

[4] JavaScript events are called mutation events if they modify the DOM tree, e.g., by changing attributes of a node, such as the *src* attribute of a *script* tag, or inserting or removing elements from the DOM tree.

7.1 Detection of Malicious Code

Numerous papers have been published on detecting malicious activity in web sites. The majority of them focus on dynamic analysis of JavaScript in instrumented environments or on rendering web sites in high-interaction client honeypots. Generally, it is important to recall that our system provides additional information: the infection campaign and the responsible node of the DOM tree.

Eshete et al. [40] discuss the effectiveness and efficacy issues of malicious web site detection techniques. Approaches from blacklists, to static heuristics, to dynamic analysis are compared in their detection accuracy and time spent analyzing the web site. A major argument in short-comings of previous work is the missing discussion on the necessity of episodic re-training or online learning capabilities, to keep up with the ongoing evolution of web-based malware, and to prevent the evasion of deployed detection systems.

Cova et al. [12] introduce the system *JSAND* to detect and analyze drive-by download attacks and malicious JavaScript in an instrumented environment. The system leverages a comprehensive dynamic analysis approach by instrumenting JavaScript to extract a variety of different features from redirection and cloaking, to deobfuscation, to observing heap exploitation. The system is compared to client honeypots, such as Capture-HPC and PhoneyC, as well as the anti-virus engine ClamAV. It shows a much lower false positive (0%) and false negative rate (0.2%) than all other approaches (5.2% to 80.6% respectively), while taking an average of 16.05 seconds to analyze a web site. CaptureHPC, the closest system in terms of accuracy takes 20 seconds per sample.

Canali et al. [13] extend the dynamic analysis system *JSAND*, by implementing a faster pre-filtering step. The main goal is to prevent the submission of certainly-benign web sites to the dynamic analysis system and hereby to reduce the time spent on analyzing benign samples, i.e., the system assigns to a false negative a much higher cost than it does to a false positive. The filter method leverages a C4.5 (J48) decision tree and a diverse set of features spanning from the HTML content, to the JavaScript code, to information about the host, to uniform resource location (URL) patterns. The filter is evaluated on a dataset of 15,000 web sites and compared to similar methods by Seifert et al. [31] and Ma et al. [39]. Both other methods yield more false positives and false negatives, but process up to 10 times more samples in the same time.

Provos et al. [11, 14] introduce a system to detect URLs to malicious web sites. However, they are not considering legitimate, infected web sites in general, as their approach is restricted to detecting the inclusion of exploit pages, and hence their approach is complementary to our system's capabilities. Their system uses a proprietary machine learning algorithm to classify URLs based on features like use in "out of place" inline frames, obfuscated JavaScript, or links to known malware distribution sites. Besides detecting 90% of all malicious landing pages with 0.1% false positives, they validate previous work by Moshchuk et al. [41] that infection vectors are inserted into legitimate web sites through exploiting vulnerabilities, advertisement networks, and third party widgets.

7.2 Web Dynamics in Security

Maggi et al. [10] introduce a web application intrusion detection system, which is able to learn about changes made to the web application. The problem of web application concept drift is addressed by learning how the web application is accessed by a legitimate user and employing an unsupervised classification algorithm. Features include, for example, a sequence corresponding to the order in which web sites are accessed or how web page parameters are distributed. However, the presented technique is orthogonal to our approach. The main goal is not to find new infection campaigns or to protect the visitor of a web site, but rather to protect the integrity of the web application. Protecting a normal, wandering user would require intrusion detection and protection of all web sites the user visits, since the access pattern, on which the system is based on, depend on the underlying architecture of the web site. Although possible theoretically, it is practically impossible.

Davanzi et al. [42] studied a similar approach for detecting the impact of web dynamics. They introduce a system to detect if changes made to a web site are defacements, which might cause serious harm to the organization, money- or reputation-wise, or are legitimate, "officially approved" content changes. However, they explicitly point out that their approach does not work with malicious modifications because their approach detects changes that are visible to the end-user, which is the exact opposite of how malicious infection vectors are placed in practice. In detail, they employ anomaly detection to regularly visit and monitor a set of 300 web sites actively and detect if changes made to the web site constitute a defacement or not.

8 CONCLUSION

In this paper, we introduced the Δ-system, a novel, lightweight system to identify changes associated with malicious and benign behavior in web sites. The system leverages clustering of modification-motivated features, which are extracted based on two versions of a web site, rather than analyzing the web site in its entirety. To extract the important modifications accurately, we introduced a fuzzy tree difference algorithm that extracts DOM tree nodes that were more heavily modified, discarding changes in single characters or words, or legitimate evolutions. Beyond detecting if a change made to a web site is associated with malicious behavior or not, we showed that the Δ-system supports the detection of previously-unknown infection campaigns by analyzing, unknown trends and measuring the similarity to previous, known infection campaigns. Furthermore, we showed that the system can generate an identifying signature of observed infection campaigns, which can then be leveraged to protect users via content-based detection systems or as test-cases for online analyzer systems. Ultimately, the system's ability to identify specific infections is helpful in identifying the reason why the web site was infected by a specific campaign in the first place, such as a distinct version of the web application among all infections; additionally, it facilitates the removal of malicious code and the mitigation of additional infections in the future.

Acknowledgment

We want to express our gratitude toward our shepherd Matt Bishop and the reviewers for their helpful feedback, valuable comments and suggestions to improve the quality of the paper.

This work was supported by the Office of Naval Research (ONR) under grant N000140911042, the Army Research Office (ARO) under grant W911NF0910553, the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537, and Secure Business Austria.

9 REFERENCES

[1] SOPHOS Security Team, "SOPHOS Security Threat Report 2013," SOPHOS, Tech. Rep., 2013. [Online]. Available: http://goo.gl/YuW65

[2] P. Baccas, "Malware injected into legitimate JavaScript code on legitimate websites," Article, 2013. [Online]. Available: http://goo.gl/rDFZ4

[3] D. Goodin, "Twitter detects and shuts down password data hack in progress," February 2013. [Online]. Available: http://goo.gl/YwfMd

[4] Facebook Security Team, "Protecting People On Facebook," Article, February 2013. [Online]. Available: http://goo.gl/OUPtk

[5] J. Finke and J. Menn, "Exclusive: Apple, Macs hit by hackers who targeted Facebook," *Reuters*, February 2013. [Online]. Available: http://goo.gl/fzhIo

[6] D. Fetterly, M. Manasse, M. Najork, and J. Wiener, "A large-scale study of the evolution of web pages," in *Proceedings of the 12th International Conference on World Wide Web*, ser. WWW '03. ACM, 2003, pp. 669–678.

[7] B. A. Huberman and L. A. Adamic, "Evolutionary Dynamics of the world wide web," *Condensed Matter*, January 1999.

[8] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. Mogul, "Rate of Change and other Metrics: a Live Study of the World Wide Web." in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, vol. 119. USENIX Association, 1997.

[9] R. Baeza-Yates, C. Castillo, and F. Saint-Jean, "Web Dynamics, Structure, and Page Quality," in *Web Dynamics*. Springer-Verlag, 2004, pp. 93–109.

[10] F. Maggi, W. Robertson, C. Kruegel, and G. Vigna, "Protecting a Moving Target: Addressing Web Application Concept Drift," in *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, ser. RAID '09. Springer-Verlag, 2009, pp. 21–40.

[11] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu, "The ghost in the browser analysis of web-based malware," in *First Workshop on Hot Topics in Understanding Botnets*, ser. HOTBOTS '07. USENIX Association, 2007, pp. 4–4.

[12] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious JavaScript code," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW'10. ACM, 2010, pp. 281–290.

[13] D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler: a fast filter for the large-scale detection of malicious web pages," in *Proceedings of the 20th International Conference on World Wide Web (WWW '11)*, ser. WWW '11. ACM, 2011, pp. 197–206.

[14] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose, "All your iFRAMEs point to Us," in *Proceedings of the 17th USENIX Security Symposium*, ser. SEC'08. USENIX Association, 2008, pp. 1–15.

[15] P.-M. Bureau, "Linux/Cdorked.A: New Apache backdoor being used in the wild to serve Blackhole," April 2013. [Online]. Available: http://goo.gl/g2Vfl

[16] D. Cid, "Apache Binary Backdoors on Cpanel-based servers," April 2013. [Online]. Available: http://goo.gl/BXq8Q

[17] S. S. Chawathe and H. Garcia-Molina, "Meaningful Change Detection in Structured Data," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. MOD'97. ACM, 1997.

[18] Y. Wang, D. J. DeWitt, and J.-Y. Cai, "X-Diff: An effective change detection algorithm for XML documents," in *Proceedings of the 19th International Conference on Data Engineering*, ser. ICDE '03. IEEE, 2003, pp. 519–530.

[19] H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[20] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation*, vol. 3, no. 0, pp. 91 – 97, 2006, the Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06).

[21] M. A. Jaro, "Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida," *Journal of the American Statistical Association*, vol. 84, no. 406, pp. 414–420, 1989.

[22] A. N. Kolmogorov, "Three approaches to the quantitative definition of information," *International Journal of Computer Mathematics*, vol. 2, no. 1-4, pp. 157–168, 1968.

[23] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions," in *Proceedings of the 19th ACM Conference on Computer and Communications Security*, ser. CCS '12. ACM, 2012.

[24] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth *et al.*, "Knowledge Discovery and Data Mining: Towards a Unifying Framework." *Knowledge Discovery and Data Mining*, vol. 96, pp. 82–88, 1996.

[25] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, "Advances in knowledge discovery and data mining," 1996.

[26] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "OPTICS: Ordering Points To Identify the Clustering Structure," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. MOD'99. ACM, 1999, pp. 49–60.

[27] M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "OPTICS-OF: Identifying Local Outliers," in *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[28] L. Invernizzi, P. Milani Comparetti, S. Benvenuti, M. Cova, C. Kruegel, and G. Vigna, "EvilSeed: A Guided Approach to Finding Malicious Web Pages," *Security and Privacy (SP), 2012 IEEE Symposium on*, vol. 0, pp. 428–442, 2012.

[29] P. Ratanaworabhan, B. Livshits, and B. Zorn, "Nozzle: A defense against heap-spraying code injection attacks," in *Proceedings of the 18th USENIX Security Symposium*, ser. SEC'09. USENIX Association, 2009, pp. 169–186.

[30] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "ZOZZLE: fast and precise in-browser JavaScript malware detection," in *Proceedings of the 20th USENIX Security Symposium*, ser. SEC'11. USENIX Association, 2011, pp. 3–3.

[31] C. Seifert, I. Welch, and P. Komisarczuk, "Identification of Malicious Web Pages with Static Heuristics," in *Telecommunication Networks and Applications Conference*, ser. ATNAC '08, 2008, pp. 91–96.

[32] "Discuz!" Retrieved, May 2013. [Online]. Available: http://goo.gl/e8nCD

[33] X. Yang, "Report on the success of the Discuz! software," Chinese National Radio Report, August 2010, in Chinese. [Online]. Available: http://goo.gl/beq4O

[34] M. Rajab, L. Ballard, N. Jagpal, P. Mavrommatis, D. Nojiri, N. Provos, and L. Schmidt, "Trends in Circumventing Web-Malware Detection," 2011.

[35] J. Mason, S. Small, F. Monrose, and G. MacManus, "English shellcode," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. ACM, 2009, pp. 524–533.

[36] M. Polychronakis, K. Anagnostakis, and E. Markatos, "Network-level polymorphic shellcode detection using emulation," *Journal in Computer Virology*, vol. 2, pp. 257–274, 2007, 10.1007/s11416-006-0031-z.

[37] J. Choi, G. Kim, T. Kim, and S. Kim, "An Efficient Filtering Method for Detecting Malicious Web Pages," in *Proceedings of the 13th International Workshop on Information Security Applications*, 2012.

[38] Y.-T. Hou, Y. Chang, T. Chen, C.-S. Laih, and C.-M. Chen, "Malicious web content detection by machine learning," *Expert Systems with Applications*, vol. 37, no. 1, pp. 55 – 60, 2010.

[39] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond blacklists: learning to detect malicious web sites from suspicious URLs," in *Proceedings of the 15th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ser. KDD '09. ACM, 2009, pp. 1245–1254.

[40] B. Eshete, A. Villafiorita, and K. Weldemariam, "Malicious website detection: Effectiveness and efficiency issues," in *First SysSec Workshop*, ser. SysSec. IEEE, 2011, pp. 123 –126.

[41] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy, "A Crawler-based Study of Spyware in the Web," in *Network and Distributed System Security Symposium*, ser. NDSS '06, 2006.

[42] G. Davanzo, E. Medvet, and A. Bartoli, "Anomaly detection techniques for a web defacement monitoring service," *Expert Systems with Applications*, vol. 38, no. 10, pp. 12 521–12 530, Sep. 2011.