# Towards Sound HTTP Request Causation Inference

*Kristin Stephens*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-141
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-141.html

August 12, 2013

Acknowledgement

We would like to acknowledge the great help many have given in this project. Vern Paxson, who advised this work and guided the writing of this report. Mobin Javed, who also worked on Gretel while this was a class project. David Wagner, who served as our second reader and also guided the writing of this report. Justin Samuel, who kindly let us use his RequestPolicy code and answered questions as it was altered for our needs. And last, but not least, my husband Chris Martinez, who supported me through the highs and lows of this work.

# Towards Sound HTTP Request Causation Inference

Kristin Stephens-Martinez

## Abstract

Network traces are a useful tool in understanding how users navigate the web. Knowing the sequence of pages that led a user to arrive at a malicious website can help researchers develop techniques to prevent users from reaching such sites. Nevertheless, inferring sound causation between HTTP requests is a challenging task. Previous work often inferred these relationships without proper calibration. We present here methods for and considerations when inferring causation relationships between HTTP requests. We also introduce causation trees and terminology needed to model causal relationships between HTTP requests. Finally, we describe Gretel, our system that infers causation relationships, how we calibrated it, and our results on a sample control data set where ground truth was available.

## 1 Introduction

When a user clicks on a link, their browser generates multiple HTTP requests, which each potentially cause even more requests. This cascade of HTTP requests merges with other cascades to give us the network traffic we see. However, which HTTP request caused another to appear is not readily apparent.

If we can infer the causation between HTTP requests there are numerous applications for both researchers and practitioners. For example, we can follow a user's progression through a single domain or identify how a user arrived at a new domain. Also, grouping the HTTP requests causally linked together lets us understand user characteristics within an Internet session, such as random jump frequency.

Causal inference can also help analyze website efficiency by allowing investigators to observe the cascade of requests caused by a single action. Observing patterns such as long sequences of HTTP requests "daisy chained" together can provide a root cause diagnosis for slow page load times.

Understanding the causal tree also has security implications. An administrator attempting to create sound security policies, such as "no files of a particular type," can apply such a policy over a network trace, removing requests not allowed by the policy. They can then display the result as it would appear when following the proposed policy.

Finally, inferring HTTP request causation can aid in forensics. For example, by differentiating between voluntary and automatic HTTP requests, we can analyze the recording of a user web attack to understand if the attack was due to a user action and, if so, which one.

The process of inferring causal relationships from a network trace, however, is not straightforward. Previous work [14, 11, 10] attempted to infer the relationship between HTTP requests using the HTTP Referer header [sic]. Few such systems explain in detail how exactly their implementation uses the referer to infer causal relationships. Even fewer calibrate their system by comparing its inferred HTTP request causal relationships against the corresponding known ground truth causal relationships. Drawing conclusions without a thorough understanding of the referer and without proper calibration can lead to a potentially skewed understanding of the network traffic.

In this work, we define the meaning of causality between HTTP requests and explain how finding causality requires more than simply looking at the referer. We present considerations required to infer HTTP request causal relationships, when using only the reconstructed HTTP messages within the network traffic and without considering the Internet objects transferred. Using these insights, we design and implement a system to infer causal relationships, dubbed Gretel, which we calibrated and ran over a sample control data set.

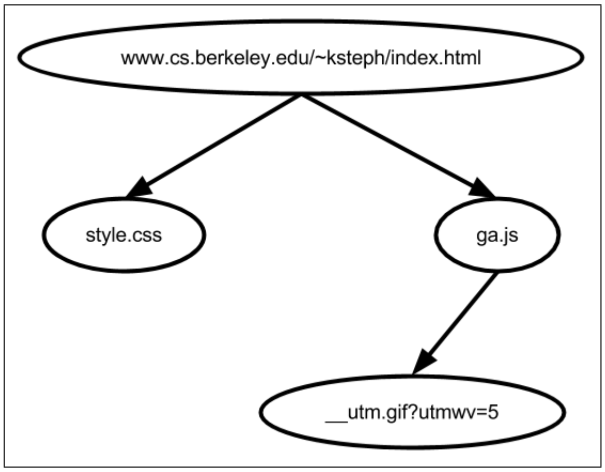


Figure 1: Small example of an HTTP request causation tree.

Our method for studying HTTP causality is minimally invasive. We record at the network layer rather than at the browser and/or server end. Using only HTTP messages without their payload helps preserve user privacy. We believe this is the right trade-off to select when compared to the information we might get from a browser-based technique. While the browser technique would provide more detailed data it is at the cost of user privacy and installation overhead.

The rest of this paper is organized as follows. In Section 2 we present causation trees and related terminology. In Section 3 we discuss previous work, and in Section 4 we cover the needed steps and considerations that we propose for inferring HTTP request causation. In Section 5 we introduce our implementation to infer causation trees, Gretel, and our calibration system. In Section 6 we describe our sample data set, and in Section 7 we compare Gretel's analysis against ground truth, providing illustrative examples of why proper system calibration provides strong results. Finally we conclude in Section 8 and discuss possible future work in Section 9.

## 2 Causation Trees

Before going further into how to infer causal relationships, we first present concepts and terminology to better discuss these relationships. In this section we introduce causation trees and new terminology defined in Table 1. We define a *causation tree* as a tree where each node represents an HTTP request and an edge connects two nodes that have a causal relationship. A *causal relationship* exists between a parent and child node when a child node would not exist if the parent did not exist. A causal relationship is represented by an edge that points from a parent node to its children. Because every node can only have one parent and a child node cannot be the cause of one of its ancestor nodes, the nodes and edges combine into a tree, or in our case a causation tree.

Figure 1 shows a small example of this. When clicking on a browser bookmark to go to the author's website, first the browser requests `index.html`. Next the browser requests `style.css` and `ga.js` (Google Analytics Javascript file) because `index.html` uses them. The `index.html` request is the parent of the `style.css` and `ga.js` requests because if it did not exist the other two requests would not exist. Finally `__utm.gif?utmwv=5` is requested due to `ga.js`, which gives us another parent-child or causal relationship.

An edge representing a causal relationship has two types, *automatic* and *voluntary*. In Figure 1 all the edges are automatic because between the request for the parent node and the child node no action was needed by the user to create the child HTTP request. This is because browsers commonly automatically load Javascript and CSS files at page load time. If there was a node representing the root's parent, in this example a bookmark, the edge connecting the two nodes would be a voluntary edge because a user action was required to request the parent node.

Nodes connected to their parent by an *automatic edge* are requests usually created by the browser or other applications that use HTTP. These requests are often needed to build the entire web page beyond its first set of HTML. They commonly include CSS, flash, xml, json, HTML, Javascript, and image files. Unless the user explicitly configured their browser to ask [15], these requests are always sent automatically without any intervention by the user. Besides components for a web page, automatic requests include security certificates, requests by non-browser programs that use HTTP to communicate over the Internet, automatic checks for updates in web applications like email, and requests that notify the service provider of a

| **Term** | **Definition** |
|---|---|
| *Node* | Represents an HTTP request and additional information found in its HTTP response. |
| *Parent* | The node that caused another node. |
| *Child* | The node caused by another node, its parent. |
| *Edge* | A directed edge that points from a parent node to a child node. |
| *Lost Child* | A node with an unknown parent. The node implies the existence of a parent through our edge detection heuristics but the parent HTTP request is not detected. |
| *Orphan* | A node that does have a parent but we are unable to find evidence of a relationship. So the node appears to have no parent HTTP request. |
| *Root* | An orphan that does not have an HTTP request as a cause. Can also be called a "true orphan." |

Table 1: Definition of causation tree terminology

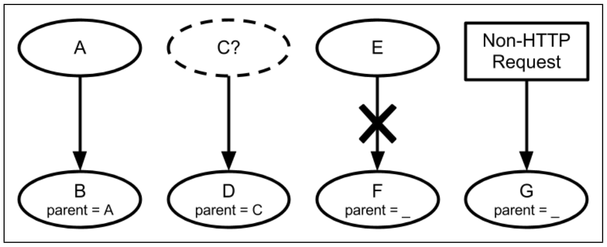


Figure 2: Different types of nodes in terms of a node's relationship with its parent. Node B is the typical parent-child relationship. Node D is a lost child. Node F is an orphan. Node G is a root.

user's activity on the service's application.

One other kind of automatic request is HTTP redirects. An HTTP redirect occurs when a requested Internet object is not where the browser originally requested it. The server's response to the browser's request includes where to look for the object instead, which is usually requested automatically.

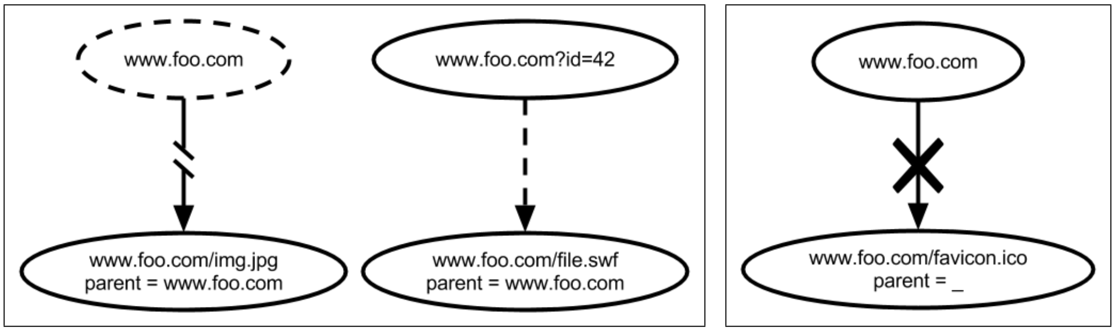
A *voluntary edge* connects two nodes where the child node exists due to a direct user action. Its existence is because of both the parent node and the user action. The common case is when the user clicks on a link in a web page. However it could also mean the user searched for something using the browser's toolbar, used a bookmark, typed into the address bar, or ran an application accessible through a browser and is interacting with it.

Because there may be gaps in our inferences, due to lack of data or imperfect inferences, we also require terminology to describe nodes that are missing a parent. The main distinction between different situations is whether a parent exists and/or there is information on who the parent is. This terminology we also define in Table 1 and show a graphical representation of in Figure 2. In the figure the first pair of nodes' relationship is the parent and child relationship we have already discussed.

The second example shows Node D as a *lost child*. A lost child does not have a parent node, but it does have information on who its parent should be. In the example our causation heuristics say Node C is Node D's parent but Node C cannot be found. This can be because Node C is not in the data set or the name Node D is using for Node C is not what Node C calls itself. A toy example can be seen in Figure 3(a). The left example shows the child node as lost because its parent is not in the data set. The right example is where the parent is in the data set, but the child has an incorrect URL to identify the parent. See Section 7.3 for real world examples.

The next example in Figure 2 with Nodes E and F represents an *orphan* relationship, where the child, Node F, does not know that its parent is Node E, but it does actually have a parent. This is because the parent could not be identified with the used heuristics. Figure 3(b) shows a toy example, where the child node is an orphan and therefore does not know who its parent is, but it does have a parent. The relationship is just missing. See Section 7.4 for real data examples.

Finally the example with Node G is of a *root*, or "*true orphan*," where Node G's parent could not be identified because its parent is not an HTTP request and therefore is not represented by a node. Such parents can be a bookmark or an application sending an HTTP request.



(a) Lost Child Examples

(b) Orphan Example

Figure 3: Toy examples of nodes as a lost child or orphan. 3(a) is two ways a node can be lost. On the left is because the node's parent is not in the data set, possibly due to the trace not starting early enough in time. The node on the right is lost because the name the node has for its parent's name is not correct. 3(b) is an orphan node because the node does not have a URL to identify its parent but it does have a parent.

# 3 Related Work

Prior work used two methods to infer relationships between HTTP requests. The more common method [14, 11, 10] is with the HTTP *Referer header* [sic], the URL of the resource from which the requested web object's URL was obtained [9]. Another method is to reduce the problem by downloading a single web page and consider the original web page HTTP request as the cause of all resulting HTTP requests [6].

Works that used the referer header usually lack an explanation on how the referer URL is matched to a requested URL [14, 10]. Explaining their process is important. Due to a URL's flexibility, there can actually be many different URL strings that point to the same web object (Section 4.1). The meaning of the lack of a referer is also usually interpreted as a new web page without any investigation to the accuracy of such an assumption [14, 11, 10]. We found that this is often untrue (Section 6.4, Section 7.4), which could actually skew the interpretation of their results. Finally some prior work that use the referer technique do not calibrate their accuracy [14, 11, 10]. Without calibration the accuracy of their inferences cannot really be judged (Section 4.2).

Qiu et al. [14] introduced the idea of a *referer tree*, where nodes represent HTTP requests and edges point from a parent node to a child node. They interpreted the child node's referer URL as the parent node's URL. Their goal was to determine the mount of web traffic influenced by search engines, as well as estimate web modeling parameters such as the number of links followed before a "random" page jump. Qiu et al. focused on only text documents, filtering on the Content Type HTTP header, and kept only HTTP requests from well-known browsers, such as "Mozilla" [2], by filtering on the User-Agent HTTP header.

Using their referer trees they found 13.6% of nodes in their trees have a search engine for a parent or ancestor. Qiu et al. also measured referer tree size, depth, and branching factor, mapping these measurements to user characteristics of page visits before a random jump, how deeply a user follows hyperlinks, and number of links clicked per page. They found all three characteristics follow a power-law curve when comparing the number of trees versus a particular characteristic. They also noted that 45% of their trees were a single node, interpreting this as a user jump with no further links clicked. However, contrastingly we found evidence that a majority of single nodes are most likely HTTP requests missing their referer and are unlikely to be a user click (Section 7.4).

Also Qiu et al. do not explain how they match referer URLs with requested URLs, simply stating the URLs are equal. There is no mention of calibrating their system either.

Meiss et al. [11] also used the referer to infer relationships between requests. They then used this information to create a host graph of domain names. Using data from the Indiana University network they looked at the difference between the traffic-weighted host graph versus the link-weighted host graph, as well as the difference between the web traffic's and PageRank's [12] host graph ranking. They found the weights in the traffic-weighted host graph are much more diverse than the link-weighted graph and there are very different rankings between the web traffic and PageRank's host graph rankings. Meiss et al. labeled each HTTP request as a human click if it is from a common browser and the URL is likely an actual web page, such as ending with HTML and not something like a style sheet or media file. Of their human HTTP requests they found 54% had no referer, which they interpreted as a page visit without clicking on a link and therefore a user jump. Once again this could be due to a missing referer.

Meiss et al. do explain how they use the referer header to match the referer URL to the request URL. For each request they first convert the URLs to the fully qualified domain name of the servers. They then find a node's parent using this domain name. While Meiss et al. do explain how they used the referer, they do not calibrate their system with a ground truth set of known relationships. They did do a sanity check on their resulting host graph, confirming it exhibited similar well known topological characteristics found by large-scale crawl studies [3, 5, 8, 16].

Ihm et al. [10] introduced their page detection algorithm StreamStructure. First, it uses the referer field to group requests into streams. Then, it uses content type and timing to determine the main objects. All objects between two main objects are grouped as part of the web page of the earlier main object. Finally, they identify the initial pages using idle times between requests of 0.5 to 2 seconds. The idle time thresholds where found by using the Google Analytics Javascript beacon, sent when the `DOMContentLoaded` event fires, meaning the web page is successfully loaded.

Ihm et al. do not include in their report how they use the referer, only saying they find a node's parent if the referer and request URL match. They also do not go into detail of how they interpret requests with no referer nor how they handle requests that are not mentioned in the referer header of any other request. Ihm et al. validate their algorithm using manually generated data by visiting and clicking through the top 100 sites of Alexa's list [1] with Microsoft Internet Explorer. Their *precision*, number of correct web pages found divided by total web pages found by the algorithm, and *recall*, number of correct web pages found by the algorithm divided by number of correct web pages browsed, where both over 0.8. This is still not the same as calibrating their inferred relationships because they only considered their validation at the level of a web page, not HTTP requests.

Butkiewicz et al. [6] looked at the complexity of individual websites by recording the HTTP requests generated when visiting the landing page of popular websites. They recorded the generated web traffic with a browser extension. From this they characterized each page by the number of objects fetched, object size, and content type. They found the main indicator of client perceived lag is the number of web objects fetched. Butkiewicz et al.'s work can be extended by looking at client perceived lag at a finer granularity. For example the lag's cause could be the daisy chaining of multiple requests which would make the fetching of some web objects to be well after others.

# 4 Needed Steps for Sound Causation Inference

## 4.1 URL Sanitization

Our primary means of identifying nodes and linking nodes together are URLs. However, URLs cause a lot of trouble because of their great flexibility. This flexibility requires converting them from their raw network traffic format to a standard format. This enables us to compare URLs between HTTP requests, using simple string comparisons. The following are our steps to convert raw URLs into a standard format.

First, all URLs are fully unescaped. URLs can be inconsistently escaped by browsers. In Table 2, URL $A_1$ is a URL requested by the browser, while URL $A_2$ is the same URL in the referer header of another HTTP request. A common trend seen in our sample data set is escaping the referer URL one more time than the request URL; see Section 6 for more details. The most levels of escapes we observed for a URL was 3, for example in Table 2 URL

| | |
|---|---|
| $A_1$ | en.wikipedia.org/wiki/Pascal's_triangle |
| $A_2$ | en.wikipedia.org/wiki/Pascal%27s_triangle |
| $B_1$ | creativeby1.unicast.com/script/V3.00/deliver2.html?exp='043012'&png=<br>'http://ping1.unicast.com/adstracking.gif&pip=<br>'http://ad.doubleclick.net/imp;<br>˜cs=g?http://s0.2mdn.net/dot.gif?1333487226525' |
| $B_2$ | creativeby1.unicast.com/script/V3.00/deliver2.html?exp=%27043012%27&png=<br>%27http%253A//ping1.unicast.com/adstracking.gif&pip=<br>%27http%253A//ad.doubleclick.net/imp<br>%257Ecs=g%25253fhttp%253A//s0.2mdn.net/dot.gif%3F1333487226525%27 |
| $C_1$ | platform.twitter.com/widgets/tweet_button.1329950604.html#_=13300172879 |
| $C_2$ | platform.twitter.com/widgets/tweet_button.1329950604.html |
| $D_1$ | pixel.quantserve.com/pixel;**[…]**;ogl=title.The Protégé Effect**[…]** |
| $D_2$ | pixel.quantserve.com/pixel;**[…]**;ogl=title.The Prot\xc3\xa9g\xc3\xa9 Effect**[…]** |

Table 2: Examples of different URLs. Text in bold represents URL portions removed for brevity.

$B_1$ is the fully unescaped URL while URL $B_2$ is the original raw URL. For easier reading each line of URL $B_1$ matches with the corresponding line in URL $B_2$.

Our next step is to remove the URL fragment, if present. The fragment starts with the "#" character and goes to the end of the URL. Request URLs can have fragments but not referers [9]. Therefore we remove the fragments for the sake of consistency and to match request URLs to referer URLs. URL $C_1$ and URL $C_2$ in Table 2 show an example of this.

Our final two sanitization steps are to encode UTF-8 characters into ASCII and replace newlines with "ˆJ". URL $D_1$ and URL $D_2$ in Table 2 show the actual and UTF-8 encoded URL respectively. WE chose to replace newlines with "ˆJ" because we use Bro 1.5 [13], which represents newlines with "ˆJ". Further Bro implementation details are in Section 5. Our main reason for these two steps is for greater consistency between the network traffic data set and the ground truth data set, which we extracted with different tools, Bro and Firefox respectively.

## 4.2 Calibration

Calibration helps us better understand the accuracy of a system's HTTP request causation inferences. Calibration means taking the output of a system's inferred causal relationships and comparing it to its corresponding known causal relationships, or ground truth. In our case we used a browser plugin for Mozilla Firefox [2] to collect our ground truth; see Section 5.3 for more details. Calibration is also useful because it provides clues on how to improve the system.

## 4.3 Other Pitfalls

Besides the steps discussed above, we found other issues we needed to keep in mind when inferring causation relationships. Included here are caveats that can occur but were not significant enough to warrant their own section.

- A user will not necessarily have the same IP address throughout the entire trace. For example, a user can change between Ethernet and wifi in the middle of their session. This complication means we could not fully rely on the TCP four-tuple to identify what node a tree belongs to because the source or destination IP address may not be the same for all the nodes in a tree.

- Useful information for a request can (and most likely will) be in both the HTTP request and response message (*e.g.,* redirect information is in the response). Measurement loss can complicate this. A lost packet may garble a message, which could hinder identifying the request and response message pairs in a TCP stream.

# 5 Implementation

Considering the steps and caveats above we implemented our own HTTP request causation inference system, Gretel. We used Bro version 1.5.3 [13] to extract information from our Internet traces. All other processing used Ruby 1.8.7. We used the Common Gateway Interface (CGI) ruby library to unescape URLs, a needed step explained in Section 4.1.

## 5.1 Information Extraction

Table 3 lists the information used to create our causation trees, found in both the HTTP request and response messages. We use the timestamp for tie-breaking heuristics. The IP address and port of the source and destination are used to help identify a request's response. We explain our use of the request URL, referer URL, HTTP response requested URL, redirect code, and location in Section 5.2. The User Agent is for filtering for only Firefox nodes when we calibrate Gretel, explained in Section 6.

## 5.2 Cause URL and Edge Heuristics

We infer a causal relationship between two HTTP requests using an HTTP request's cause URL. The *cause URL* is the URL of a node's parent and identifies the parent node. To find the cause URL we first use Bro to extract the information in Table 3 and then merge the HTTP request and response information into a node. Finally we infer the causal relationships, or the edges between nodes, by using the following edge heuristics.

- **Referer Header:** This header is in the HTTP request message. Its corresponding value is the URL of the resource where the client found the requesting URL, usually called the *referer*. Its original purpose was to provide servers with information to create back-link lists [7, 9]. This header is technically optional and web clients are cautioned to be careful with private information. However it is usually sent by default, with some exceptions [4]. Therefore, the referer is an ideal candidate for determining the cause URL of a request.

  Absent more information, we use the referer as the cause URL.

- **HTTP Redirects:** These are HTTP reply messages with status codes in the 300s. They generally mean further action is needed before fulfilling the HTTP request. If the status code is 301, 302, 303, or 307, the object requested should be looked for at the URL in the HTTP Location header's value [9].

  If a request is due to a redirect, the cause URL is the URL of the HTTP request that received the redirect reply.

- **Tie-Breaker:** When multiple nodes have the same URL, there are therefore multiple parent node candidates. The candidate closest in time to the child node is chosen as the parent.

The referer header is what most prior work [10, 11, 14] used to create what Qiu et al. [14] called referer trees. However since we focus on HTTP request causation trees, using *only* referers will not create accurate causation trees. When a request is sent due to a redirect, the value of the referer header is unclear. It could be the original redirected HTTP request's referer, the URL of the redirected message, or empty. In our data set we found Firefox always set the referer as the referer of the originally redirected HTTP request; see Section 6.4 for more details. However there is no established agreement across browsers on what the referer should be for redirects.

A toy example with and without the redirect heuristic is in Figure 4. Figure 4(a) shows the series of messages. Figure 4(b) is the causation tree, where `www.foo.com/img1.jpg` is the parent of `www.bar.com/img1.jpg` because the first request was redirected to the second. Figure 4(c) is the referer tree, assuming the referer was set to the original redirected HTTP request and therefore used as the cause URL.

## 5.3 Calibration

To calibrate Gretel we need the ground truth for the network traffic we give our system. Ground truth is the actual causal relationships between HTTP requests. We used a Firefox plugin to collect our ground truth, while simultaneously recording the corresponding network traffic.

| Both | HTTP Request | HTTP Response |
|---|---|---|
| Network Timestamp | Requested URL | Requested URL (inferred by Bro) |
| Source IP address | Referer URL | Content Type |
| Source Port | User Agent | Redirect Code (if a redirect) |
| Destination IP address | | Location (if a redirect) |
| Destination Port | | |

Table 3: Information used by Gretel to create causation trees.

| Firefox Notification or Function | Documentation |
|---|---|
| `http-on-examine-request` | Notification when Firefox creates an HTTP request and provides the HTTP request content |
| `http-on-examine-response` | Notification when Firefox receives an HTTP response and provides the HTTP response content |
| `shouldLoad` | Called before loading a resource to determine whether to load it, provides both the resource and the object that wants to load the resource |

Table 4: Documentation of Firefox functions.

### 5.3.1 Firefox Plugin Implementation

We altered a preexisting Firefox plugin, RequestPolicy [15], that already tracked the cause of each request. Using its logic we recorded the information needed to create the ground truth causation trees.

The bulk of the code we inserted into RequestPolicy is in Mozilla's nsIContentPolicy's function `shouldLoad` and two Firefox plugin events, `http-on-examine-request` and `http-on-examine-response`. In other words, a node contains information found in all three function calls. Table 4 provides documentation on these three functions.

Firefox calls `shouldLoad` for all content that needs rendering on the page. It has an optional parameter, `aRequestOrigin`, that provides the location of the resource that initiated the load request. The value of `aRequestOrigin` is what we consider the cause of the request. We used the events `http-on-examine-request` and `http-on-examine-response` to see each message's content. This lets us collect the other information listed in Table 3 when comparing to Gretel's output. These events also allow us to track any redirects, which Firefox does not take into account when providing the cause URL in `aRequestOrigin`, and apply our redirect heuristic discussed in Section 5.2.

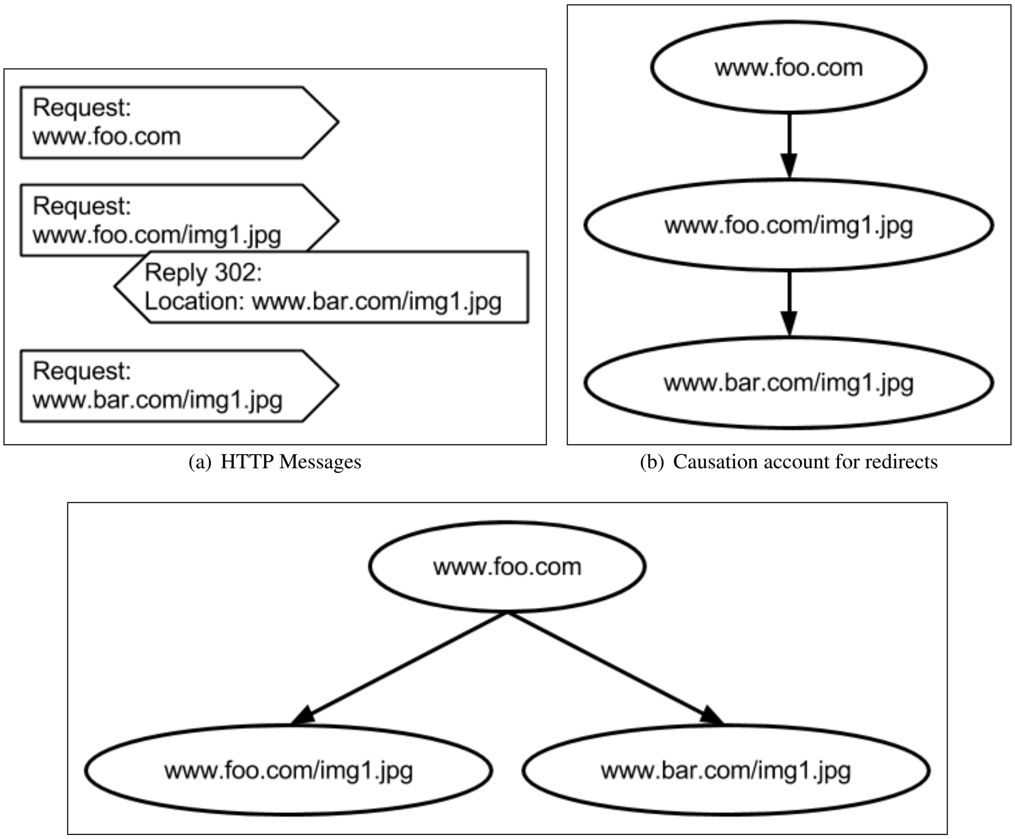
Another advantage of using a plugin is `shouldLoad` allows us to identify actual roots of a tree. Any node that Firefox identifies as having a cause URL of `chrome://browser/content/browser.xul` is a root because this URL actually means the browser itself.

While we collected our ground truth from the vantage point of the browser, we do recognize that our implementation does not fully capture what we want. At this time we are not able to actually determine the accuracy of `shouldLoad`'s optional `aRequestOrigin` parameter. We also needed to track the redirect messages outright. A more accurate ground truth collection would involve digging deeper into Firefox to find and track the formation and progression of all HTTP requests.

### 5.3.2 Complexities using Firefox Plugin

The possibility of recording an inaccurate ground truth is further compounded by complexities we found while using the plugin, which we list here.

1. Firefox may call the `http-on-examine-request` listener without actually sending the request because of browser caching. This results in a false negative when



(a) HTTP Messages

(b) Causation account for redirects

(c) Causation only using referers

Figure 4: Example of the difference between only using referers to infer causation and also including redirects.

comparing the ground truth with the output of the causation tree inference implementation. This is because the ground truth shows the existence of an HTTP request but it is not in the network traffic. This required us to set Firefox to “no caching” to avoid these false negatives.

2. Firefox may include the fragment in a URL when reporting it to our plugin, but not actually send the fragment with the URL. This is handled in our URL sanitization step that removes fragments, discussed in Section 4.1.

3. The output from Firefox is not encoded the same as Bro. Firefox provided data in UTF-8 and with newlines, while Bro only output ASCII and replaced newlines with “ˆJ”. To handle this we preprocess the Firefox output to replace all UTF-8 encoding with ASCII and replacing newlines with “ˆJ”, similar to what we mentioned in the URL sanitization step in Section 4.1.

4. Some requests are not seen in `shouldLoad`. These requests are for objects such as security certificates, which are not rendered on a web page. Further work is needed to find ways to confirm the cause of requests like these.

5. In a reversal of the previous pitfall, we also found Firefox calls the `shouldLoad` function 1.4x -

2.9x more often compared to the actual number of requests. One reason for this is a web page may render the same object multiple times, calling the shouldLoad function each time it needs to be rendered. This is different from the caching problem because no request is ever created, so http-on-examine-request is never called.

   One example of this is when we loaded maps.google.com in our data set. There were 1398 calls to shouldLoad for mapcontrols3d7.png compared to 3 actual requests.

6. shouldLoad and our listener on the http-on-examine-request event are sometimes called in reverse order. This can cause complications when linking a node's data from the shouldLoad call to the information found in the HTTP events.

The best way to handle problems 4, 5, and 6 is to log every call to all three functions.

To handle the overzealous calling of shouldLoad, which causes many "extra" nodes, we take two additional steps. First, while creating the causation trees and there are multiple parent node candidates, we always prefer a parent node that is actually seen in http-on-examine-request. Second, once we finish comparing the ground truth from Firefox and our causation inferences from Gretel, and therefore have an idea of what nodes were actually sent, we removed from consideration all nodes Gretel would not have been able to see. In particular we removed nodes Firefox found but were not actually requested, such as those seen in shouldLoad but not http-on-examine-request. We also removed any requests sent through HTTPS, identified by the scheme used in the HTTP request.

To fix the problem of reordering, we track nodes that are missing their corresponding http-on-examine-request or shouldLoad function call. We merge two nodes, and therefore consider it a reorder event, if:

1. They have the same URL,
2. They are missing the function call the other has, and
3. They are within 0.1s of each other

# 6 Data from a Controlled Environment

## 6.1 Data Set Collection

We recorded five Internet browsing sessions as a sample control data set for Gretel. Table 5 shows the basic information for each session: day of recording, interface used, duration, and number of nodes, edges, trees, lost children, and orphans in the session's data. We define a *tree* as having more than one node.

We recorded all sessions on a Lenovo T410 laptop running Ubuntu 10.04. We used Firefox 10.02 for the first session, Session A, and Firefox 11.0 for all other sessions, Sessions B through E. Most of the recorded Internet traffic included reading news and blog articles at various websites initially accessed with Google's Reader application and using YouTube.com, Google Maps, and a Flash application. The diverse sources of traffic contributed to the broad range of node and edge counts regardless of the Internet traffic recording's duration.

At this point we would like to note this data set is only a small controlled sample of network traffic and is not necessarily representative of any particular network. However it does serve as an existential proof that behavior like this does exist. Also due to the small and very diverse sample data set, unless otherwise stated we will refer to the entire data set as a whole.

We recorded each session as follows:

1. Start Internet traffic recording, with tcpdump
2. Open Firefox
3. Browse Internet
4. Close Firefox
5. Stop tcpdump

Starting the recording before the start of Firefox ensured we record all HTTP requests Firefox sent. Also after we extract our needed information with Bro, we filter on the user agent HTTP header value so we only compare requests sent by Firefox to the ground truth.

| Session | Date | Interface | Duration | Nodes | Edges | Trees | Lost Children | Orphans |
|---|---|---|---|---|---|---|---|---|
| A | 2012-02-23 | wlan0 | 1:16:48 | 1,700 | 1,457 | 10 | 1 | 242 |
| B | 2012-04-03 | eth0 | 0:51:33 | 3,224 | 3,026 | 20 | 57 | 141 |
| C | 2012-04-05 | wlan0 | 0:13:08 | 1,851 | 1,747 | 10 | 18 | 86 |
| D | 2012-04-10 | wlan0 | 0:53:31 | 11,006 | 10,410 | 40 | 67 | 529 |
| E | 2012-04-13 | wlan0 | 0:08:23 | 627 | 571 | 7 | 4 | 52 |
| **Totals** | | | **3:23:22** | **18,408** | **17,211** | **87** | **147** | **1,050** |

Table 5: Information about each data set collected.

## 6.2 Preliminary Analysis

Using the `capture-loss.bro` Bro script, we had a mean Bro event loss of 0.13% and mean byte loss of 0.70% for our network traffic.

From Table 5 we see that of our 18,408 nodes, 1,197 have no parent (6.5%). Of these parentless nodes, 147 are lost children (12.3%) and 1,050 are orphans (87.7%). 87 of the nodes with no parent have children and are therefore considered a tree. 24 of the nodes at the top of these trees are lost children and 63 are orphans.

23.4% of the nodes had multiple potential parents, with the parent chosen using our tie breaker heuristic in Section 5.2.

| | | Count of unescapes for requested URLs | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | Total |
| Count of unescapes for cause URLs | 0 | 0 | 0 | 0 | 0 |
| | 1 | 1,628 | 333 | 0 | 1,961 |
| | 2 | 0 | 657 | 0 | 657 |
| | 3 | 0 | 0 | 9 | 9 |
| | Total | 1,628 | 990 | 9 | 2,627 |

Table 6: Number of requested URLs and cause URLs connected by an edge per number of required unescapes for each URL. This table shows URLs that only matched because of sanitization.

## 6.3 Referer Complexities

We found 96.1% of the edges in our data set through referers (16,535 out of 17,211). Of these edges we needed URL sanitization, discussed in Section 4.1, to find 2,627 (15.9%). This means 15.9% of the raw referers do not match the raw requested URL they are referring to.

Of these 2,627 edges, 94.7% did not match because of inconsistent browser escaping. Table 6 shows the number of requested URLs and cause URLs on each side of an edge per number of times each had to be unescaped. Notice that there is a general trend of escaping the cause URL one more time than the requested URL.

The step after unescaping the URLs, discussed in Section 4.1, is to remove the fragment. Using this step, we found the other 5.3% of the edges, where the raw referer did not match the requested URL it was referring to. When analyzing the Gretel only data set, we did not actually need to check for the last two URL sanitization steps because Bro automatically converts UTF-8 into ASCII and replaces newlines. However, we needed the last two steps for our ground truth because Firefox output has different formatting; see Section 5.3.2 for details.

## 6.4 Redirects

676 (3.7%) of the nodes represent requests due to redirect replies. We found that some of these nodes are chained together, with the longest path containing 20 nodes. Therefore while there are 676 redirect nodes, there are only 472 paths; 16.7% of these have more than one redirected node.

We were able to find the parent for all but four of our nodes caused by redirects. Of the nodes we found parents for, their referer matched the referer of the original redirected HTTP request.

The referer matching the originally redirected HTTP request is also consistent with the 58 nodes without a referer. If we only used the referer heuristic these nodes would be considered orphans, even though we can identify their cause.

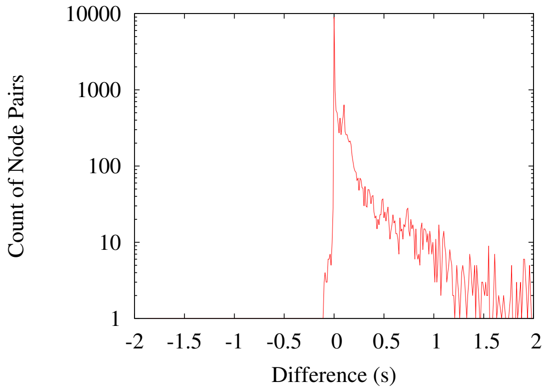


Figure 5: Histogram of timestamp difference between paired Gretel and ground truth nodes. Notice the y-axis uses a log scale.

# 7 Comparison with Ground Truth

Comparing the ground truth to the results from Gretel requires comparing two forests of trees. Unfortunately, with so many nodes and edges, ambiguities arise on how to match between the two forests. This is especially true because some nodes have the same requested URL.

In this section we first describe how we compared the two forests. Then we discuss our overall accuracy. Finally, we look at Gretel's lost children and orphans with the added context of the ground truth.

## 7.1 Implementation

We compared the two forests of trees by using a node-centered approach. First, we did this by pairing nodes from the ground truth causation tree forest to nodes in Gretel's causation tree forest. Only after we match the nodes, do we look at the edges by considering the edge that connects a node to its parent.

To pair ground truth and Gretel nodes, we used each node's meta-data. This greatly simplifies the problem, but it does not eliminate all the complexities.

Below are the steps we make when comparing the two forests.

1. For each node in the ground truth: Find all nodes from Gretel that match with the following criteria:
   (a) Request URLs are equal,
   (b) Same scheme, and
   (c) Timestamps are within 2.0s of each other
2. Tie Breaker: If there is more than one possible match:
   (a) Choose the node with the most matching children URLs
   (b) Otherwise choose the one closest in time

The same scheme condition is needed because the Firefox plugin can see both HTTP and HTTPS, however Gretel cannot and therefore has no HTTPS nodes.

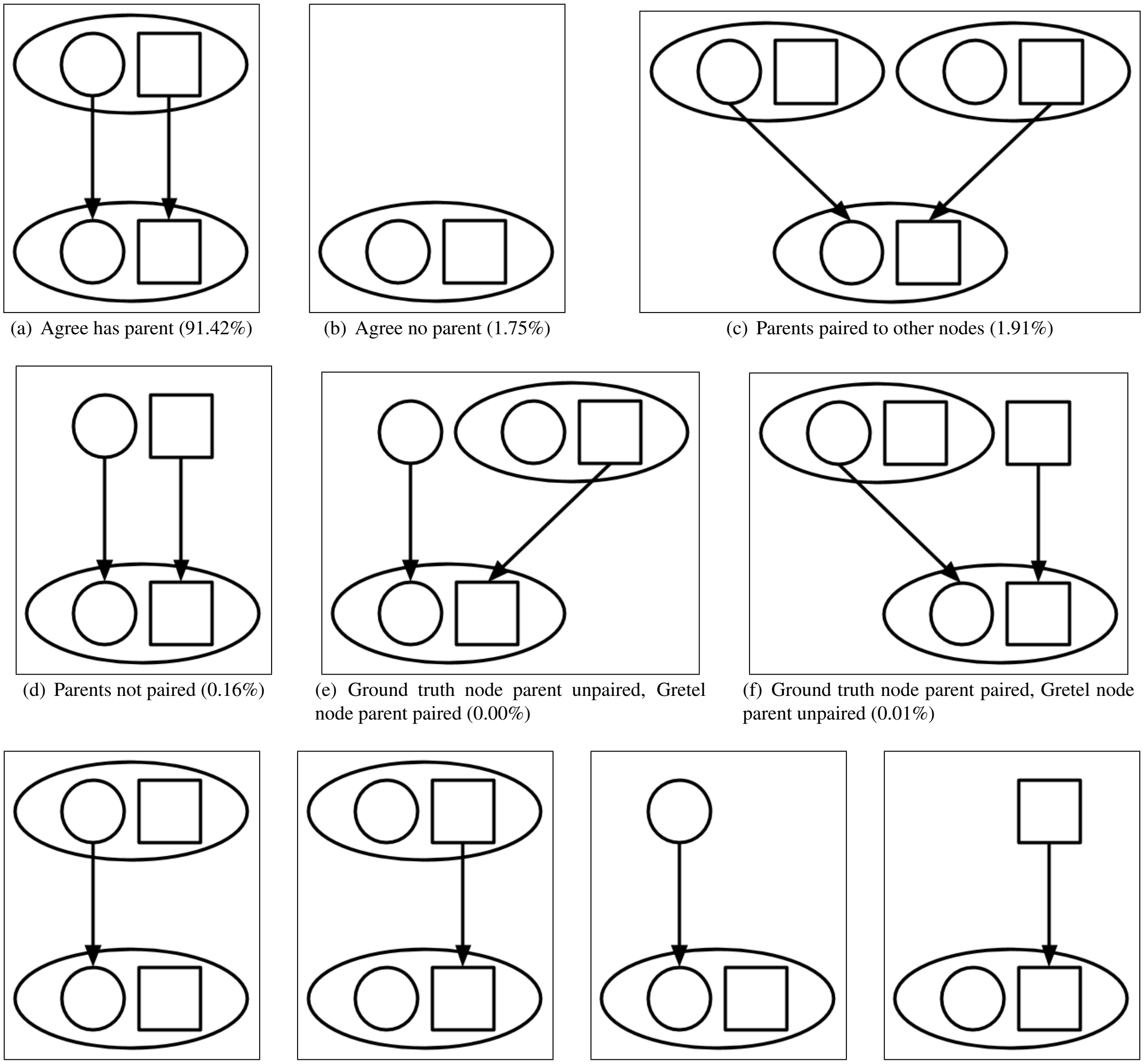
We have the timestamp difference limit because there is a slight difference in the Firefox plugin and Bro timestamps and URLs can be requested multiple times. Therefore to reduce the possibilities of incorrect node pairings we limit how far apart the nodes' timestamps can be. Figure 5 shows a histogram of the time differences between Gretel and ground truth node pairs. Note the log scaling on the y-axis. 97 (0.5%) of the node pairs had the Gretel node's timestamp before the ground truth node's timestamp. 98% of the paired nodes were within 1.0s of each other, so we considered 2.0s adequate.

We used the tie breaker to find the matching Gretel node for 8.9% of the ground truth nodes. 4.3% of these ties used the "most matching children URLs" rule, while 95.7% used time. This is not surprising because 89.2% of the nodes in our data set do not have children. Even though our system used time more often, we still want the number of matching children to take precedence over time when such information is available.

## 7.2 Overall Accuracy

Since there are two steps in our comparison, our accuracy is also in two parts. First is how many nodes were paired between the ground truth and Gretel data sets. Figure 6 shows the percentage of ground truth nodes we were able to pair with a Gretel node.

Second is how many of these pairs of nodes agree on the paired parent nodes. There are 10 possible parent child relationship scenarios for the paired nodes, shown in Figure 7. A circle represents a ground truth node, while a



(a) Agree has parent (91.42%)

(b) Agree no parent (1.75%)

(c) Parents paired to other nodes (1.91%)

(d) Parents not paired (0.16%)

(e) Ground truth node parent unpaired, Gretel node parent paired (0.00%)

(f) Ground truth node parent paired, Gretel node parent unpaired (0.01%)

(g) Only ground truth node has paired parent (4.59%)

(h) Only Gretel node has paired parent (0.05%)

(i) Only ground truth node has unpaired parent (0.12%)

(j) Only Gretel node has unpaired parent (0.00%)

Figure 7: All possible parent child relationship scenarios after pairing a ground truth node, depicted as a circle, with a Gretel node, depicted as a square. The percent of occurrences for each scenario in our dat set are in parenthesis

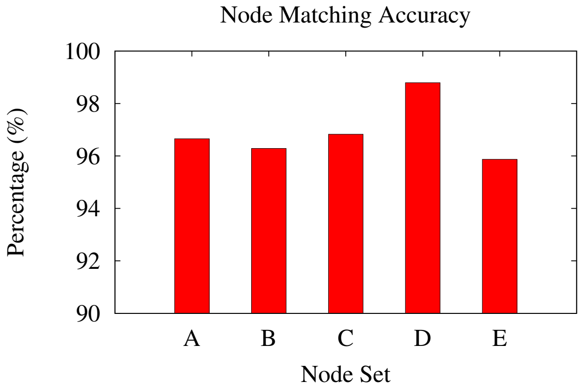


Figure 6: Percent of ground truth nodes paired with a node in Gretel's data set. Note the y-axis does not start at 0.

square represents a Gretel node. An ellipse around both means they are paired together.

Figures 7(a) and 7(b) are the two scenarios where the paired ground truth and Gretel node agree on their parent. All the other figures, 7(c) to 7(j), are how a pair of nodes can disagree. Figures 7(c) through 7(f) are when both nodes in the pair have a parent, but the parent is paired with a different node or not paired at all. Figures 7(g) through 7(j) are when only one of the nodes has a parent and that parent is either paired or not.

Within the caption of each subfigure in Figure 7 is the percent of times that scenario occurred. 93% of the paired nodes agreed on the parent or agreed there is no parent. 7% of the paired nodes disagreed. Looking further we found of the node pairs that disagreed, 64% were in scenario 7(g), where only the ground truth node has a parent and it is paired with a Gretel node. This made up 5% of all paired nodes. Also 27% of disagreeing nodes pairs were in scenario 7(c), where both nodes in the pair have a parent but those parents are paired to different nodes, which is 2% of all paired nodes.

## 7.3 Gretel's Lost Children

There are a total of 147 lost children in Gretel's causation inference of our data set, 0.8% of the total nodes. A lost child is a node with a cause URL, but we could not find the node with that cause URL. Table 7 shows the breakdown of different kinds of lost children.

| **Categories of lost children** | **Proportion** |
|---|---:|
| Flash | 81% |
| Firefox remembered tabs | 16% |
| API Call | 3% |

Table 7: Categories of lost children, sorted in descending percent order.

The majority of our lost children were requests by Flash, identified by the cause URL having ".swf" as the ending. In Table 8 URLs $A_1$ and $A_2$ are examples of URLs with Flash file URL $A_c$ as their cause URL. However there is no node with URL $A_c$ as its URL, only nodes with the same URL but no query string, like URL $A'_c$. When looking at the ground truth node paired with these lost children we found its cause URL is completely different from the Gretel node's cause URL, URL $A''_c$. We also found the ground truth node had no referer URL, but the corresponding Gretel node's cause URL came from the referer header. This is more evidence of unexpected Firefox behavior and the need for calibration. At this time we were not able to find why it ocurred.

The "Firefox remembered tabs" lost children are HTTP requests made by Firefox when loading tabs saved from the previous Firefox session. We identify them as nodes requested during the first 30 seconds of the session. An HTTP request made when loading the previous session's tabs could be considered lost because the request that would be its parent is in the previous session of Firefox, which is an unknown time before.

The last of the lost children were API calls, identified as URLs containg a "?" followed by a query string, such as example URL $B$ in Table 8. We could not identify a consistent pattern for why these nodes were lost children.

## 7.4 Gretel's Orphans

In our data set there were 1,050 orphans, 5.7% of the total nodes. Orphans are similar to lost children because they have no parent, but differ because they do not have a cause URL identifying their parent node. Table 9 shows the different categories of orphans and the percent of orphans in that category. We were able to categorize all but 3% of the orphans.

The majority of our orphans were API calls. We iden-

| | |
|---|---|
| $A_1$ | `receive.inplay.tubemogul.com/StreamReceiver/services` |
| $A_2$ | `inplay-rcv25.tubemogul.com/StreamReceiver/services` |
| $A_c$ | `static.inplay.tubemogul.com/core/core-as3-v4.5.3.swf?playerID=P-RS5-841&`**[...]** |
| $A'_c$ | `static.inplay.tubemogul.com/core/core-as3-v4.5.3.swf` |
| $A''_c$ | `www.esquire.com/features/young-people-in-the-recession-0412-3` |
| $B$ | `api.invideous.com/plugin/get_user_info?`**QUERY_STRING** |
| $C$ | `safebrowsing-cache.google.com/safebrowsing/rd/`**[...]** |
| $D_1$ | `www.google.com/favicon.ico` |
| $D_2$ | `s2.youtube.com/crossdomain.xml` |
| $E_1$ | `ocsp.entrust.net/` |
| $F_1$ | `start.ubuntu.com/10.04/`**[...]** |
| $F_2$ | `fxfeeds.mozilla.com/`**[...]** |
| $G$ | `www.google.com/search?client=ubuntu&channel=fs&q=tablet reviews&`**[...]** |

Table 8: Examples of different URLs, text in bold represent URL portions removed for brevity.

| **Categories of orphans** | **Proportion** |
|---|---:|
| API Call | 40% |
| System file | 21% |
| Firefox remembered tabs | 13% |
| Picture | 10% |
| Flash | 9% |
| Security | 3% |
| Unknown | 3% |
| Root (True Orphan) | 1% |

Table 9: Orphan categories, sorted in descending percent order.

tified them by looking for a "?" and a query string afterwards, same as lost children, and manual examination that identified common API calls. For example URL $C$ in Table 8 is one of many Google API calls used by Firefox to detect phishing. When looking at the paired ground truth node for these orphan API call nodes, some also do not have parents. This is especially true for those nodes with URLs like the phishing detection with Google's API. These nodes may be roots or true orphans, in that they do not have an HTTP request as their cause. Instead, the browser is their cause.

System files are favicon.ico and crossdomain.xml files for a website, like URL $D_1$ and $D_2$ in Table 8. Pictures are any files that ended with jpg, png, or gif. We looked at the corresponding ground truth node for these two types of nodes and found they have parents. However the referer is not set nor does this involve redirects, therefore Gretel would not be able to identify a cause URL with its current heuristics. At this time why Firefox did not set the referer header is not known.

Security nodes are those with URLs recognized as security certificates, such as URL $E_1$, identified when the URL references a domain's OCSP subdomain. The paired ground truth node for these requests also did not have a parent nor a cause URL. Why and when Firefox sends a request for a security certificate is not clear and therefore we do not know whether these requests are orphans or roots.

We identified 13 of the 1,050 orphans as roots or true orphans. A node was considered a true orphan if it was an automatic request by Firefox on startup, like URLs $F_1$ and $F_2$, or a search using the Google search toolbar in Firefox, URL $G$. We were able to confirm this by looking at the paired ground truth node and checking if its cause URL is `chrome://browser/content/browser.xul`, which is referring to the browser.

# 8 Conclusion

Previous work attempted to infer the relationships between HTTP requests, usually by using only the referer header. However previous work does not always explain how the referer is used to infer the relationships, which can affect their level of accuracy. Some also did not calibrate their system, which requries checking their inferred relationships against known ground truth. Not understanding their accuracy, potentially skews their results.

To better understand HTTP request relationships we introduced the idea of causation trees. Causation trees are a representation of Internet traffic at the level of HTTP requests. They show which HTTP request caused another to occur and provide a means to understand Internet traffic at a finer granularity than previously researched.

We then discussed an approach for sound causation tree inference, as well as potential pitfalls and complexities when creating the trees and collecting the ground truth.

Finally we presented Gretel, a first attempt at creating causation trees. When comparing its results to our ground truth using our control data set, we achieved over 95% node pairing and 93% edge agreement within our trees. However, as we said before, our data is from a small controlled sample of network traffic. Therefore our accuracy should more show the strength of our existential proofs by example that behaviors like this can happen.

# 9 Possible Refinements

At the time of this writing more can be done for inferring sound causation between HTTP requests. Below is a list of possible refinements.

- To discern between voluntary and automatic HTTP requests, use known voluntary/automatic requests to create probability distributions of the times between the request and its parent. The distribution for each possibility can then be used to label the unknown voluntary/automatic requests. The time difference between the parent and the unknown request can be compared to the distribution to find the probability it is voluntary or automatic; see Section 2 for definitions.

- Use URL parameter to help determine the parent of a node. API calls, ActiveScript, and Flash do not necessarily have the referer set, but the parameters may provide enough information. However it cannot be assumed the parameter field name is always the same (*e.g.,* YouTube.com uses a mix of docid, video_id, and v as the parameter field name holding the video's ID ).

- More information is available than what we used for Gretel. Appendix A lists all the information we recorded, with anything we did not use marked.

# 10 Acknowledgements

We would like to acknowledge the great help many have given in this project. Vern Paxson, who advised this work and guided the writing of this report. Mobin Javed, who also worked on Gretel while this was a class project. David Wagner, who served as our second reader and also guided the writing of this report. Justin Samuel, who kindly let us use his RequestPolicy code and answered questions as it was altered for our needs. And last, but not least, my husband Chris Martinez, who supported me through the highs and lows of this work.

# References

[1] Alexa the web information company. www.alexa.com.

[2] Mozilla firefox. www.mozilla.org/en-US/firefox/new/.

[3] R. Albert, H. Jeong, and A.-L. Barabási. Internet: Diameter of the world-wide web. *Nature*, 401(6749):130–131, 1999.

[4] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 75–88, New York, NY, USA, 2008. ACM.

[5] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer networks*, 33(1):309–320, 2000.

[6] M. Butkiewic, H. V. Madhyastha, and V. Sekar. Understanding website complexity: Measurements, metrics, and implications. In *In Proc. Internet Measurement Conference*, 2011.

[7] S. Chakrabarti, D. A. Gibson, and K. S. McCurley. Surfing the web backwards. *Computer Networks*, 31(11-16):1679 – 1693, 1999.

[8] D. Donato, L. Laura, S. Leonardi, and S. Millozzi. Large scale properties of the webgraph. *The European Physical Journal B-Condensed Matter and Complex Systems*, 38(2):239–243, 2004.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertex transfer protocol – HTTP/1.1, 1999.

[10] S. Ihm and V. S. Pai. Towards understanding modern web traffic. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pages 295–312, New York, NY, USA, 2011. ACM.

[11] M. R. Meiss, F. Menczer, S. Fortunato, A. Flammini, and A. Vespignani. Ranking web sites with real user traffic. In *Proceedings of the international conference on Web search and web data mining*, WSDM '08, pages 65–76, New York, NY, USA, 2008. ACM.

[12] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[13] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435 – 2463, 1999.

[14] F. Qiu, Z. Liu, and J. Cho. Analysis of user web traffic with a focus on search activities. In *In Proc. International Workshop on the Web and Databases (WebDB*, pages 103–108, 2005.

[15] J. Samuel. RequestPolicy. `www.requestpolicy.com`.

[16] M. Serrano, A. Maguitman, M. Boguñá, S. Fortunato, and A. Vespignani. Decoding the structure of the www: A comparative analysis of web crawls. *ACM Transactions on the Web (TWEB)*, 1(2):10, 2007.

# A Recorded Network Information

## HTTP Request

- Network time
- TCP:
  - Source IP address
  - Source port
  - Destination IP address
  - Destination port
- URL scheme*
- URL
- URL fragment*
- Origin*
- User agent
- Cookies*

## HTTP Response

- Network time
- TCP:
  - Source IP address
  - Source port
  - Destination IP address
  - Destination port
- URL
- Status code
- Location header value
- Content type*

*Information not used in our analysis