Intelligent Detection of Malicious Script Code

Kamron Farrokh, Benson Luk, and Eyal Reuveni

As it stands today, malicious script detection is done with a database of known attacks. As a script runs, a program checks the script's actions against this database, and if a correlation exists then the script will be labeled as malicious. There is a problem with this method, however, in that computer are vulnerable to newly created attacks, until these attacks are recorded in the database.

Our proposed method to solve this problem is to find statistical trends that exist in acceptable (i.e. non-malicious) script code, and block anything that veers from these trends. The hope is that this will any attack, regardless if it is new or old, and eliminate the need for constant updating of the database of known attacks.

Table of contents

Infrastructure

The most important element in distinguishing malicious code from non-malicious code is having a good set of data. To ensure that we were only handling good data, the URLs at which the Javascript listener is used must not contain any malicious scripts. To do this, we adopted two strategies: the first is to only visit URLs from specified (known to be safe) domains, the second is to check the log files of Canary (a Symantec product that blocks malicious scripts) and make sure that no malicious behavior has been noted while running the scripts from the site. Once it is certain that the current site is fine to examine, then the data about the script behavior is stored.

There is a component in the project that implements each of these elements. A web crawler is used to reliably grab a list of good sites. A Python script was written to run through the list of URLs and double check that the script activity is not malicious. We wrote code that embedded into Canary to extract the data from Internet Explorer. Each of these components, along with its development, is described in separate sections, as well as an appendix that will elaborate upon specifics in the implementation.

Testing Environment

All testing was done on Windows XP SP2 Professional under Internet Explorer 7. Additionally, Norton Antivirus 2008 was installed on the test machine. Not only would the product ensure that the system remained clean, but its Canary technology would enable us to know when a site attempted to infect us and, in a later stage, would allow us to hook directly into the Javascript interpreter to obtain data relevant to our research.

Web Crawler

The URL crawler chosen for this project was the open-source Heritrix, created by the Internet Archive [1]. It was chosen specifically because of the excellent documentation and customizability. There were several modifications that had to be made to get the crawler running as needed for the research. The first concern was having the crawls limited to certain domains, so that sites with malicious scripts could not accidentally be included. The second concern was the output of the crawl, as only the URL is needed at each site visited. The third concern was the depth of the crawl, as not limiting this factor would lead to endless crawls with very skewed data. These three options were available in the Heritrix crawler without modifying any of the code.

There are several modules that are contained within Heritrix that can be customized to change its behavior. One can modify the rules for which URLs are visited at several levels. The FilterScope, SurtPrefixScope, DomainScope, HostScope, and PathScope can be used to achieve the goal of our crawl, however they were all found to be inadequate in terms of the customizability that was necessary.

A module called the DecidingScope was the chosen scope for the crawl. It is much more customizable, as it can be set using several submodules. The important ones for our crawl were the OnDomainDecideRule, TooManyHopsDecideRule and MatchesFilePatternDecideRule[2]. The OnDomainDecideRule rejects sites that are not on the same domain as the initial seeds provided. The TooManyHopsDecideRule limits the depth of the crawl, and the MatchesFilePatternDecideRule prevents the crawler from visiting miscellaneous files that could not possibly contain Javascript. The order of the

scope rules is pertinent as well, since they are applied as they are listed in the submodules section of the DecidingScope. The rules can affect the decision in two ways, by supplying either a positive or a negative feedback on each URL.

The way our crawls were set up are as follows:

AcceptDecideRule	Accepts by default
NotOnDomainDecideRule	Rejects a URL that is off the seed domains
NotMatchesFilePatternDecideRule	Rejects a URL that cannot contain scripts
TooManyHopsDecideRule	Rejects a URL that is too deep

Our crawl conditions required that, logically, all three rules be passed in order for a URL to be visited, and so setting up the crawl rules this way forces every site to pass each condition.

Another requirement for the crawler was minimal storage of data. Since only the URL of every site was required, most of the data Heritrix stores by default is unnecessary. Unless the settings are changed, Heritrix stores a lot of information regarding each site using its native ARC writer, as well as a log of each site visited in a file called crawl.log. In order to minimize the amount written to the hard drive, the ARC writer was completely removed and only the log file was considered, since the log file can easily be parsed by a simple script to extract the URL for each site.

Once these settings were changed, all that was needed to run a crawl was to simply enter the seeds. These were chosen from the Alexa 10,000 list, which contains the 10,000 most popular sites on the web. The list is ranked based on popularity only, so certain sites were not considered for our crawls – instead, domains were picked based on whether we had heard of the site before. These are the crawls we ran using the settings:

Crawl	Domains	Time	Depth	URLs(approx)
1	Google, Yahoo, MSN, Youtube, MySpace	9 days	5	6,500,000
2	Google, Yahoo, MSN, Youtube, MySpace, Facebook, Live.com, Wikipedia, Ebay, Amazon, Orkut,	5 days	3	3,000,000
3	200 popular domains	3 days	2	3,000,000
4	200 popular domains	1 day	1	18,500

An issue that arose with the crawling was that Heritrix creates very large state files to hold its queues, and these could not be stored in the 20 GB hard drive that was originally on the computer. The problem arises in that Heritrix handles URLs with a number of threads (which are referred to as ToeThreads in the Heritrix lingo), and each has its own queue. When a crawl is particularly entangled and/or deep, one or two queues may fill up significantly. The URLs stored in those queues do not spill over to less-filled threads, resulting in a large state (up to 10 GB before crashing). To alleviate this, a 750 GB hard drive was purchased and installed, which allows Heritrix to run freely. The available disk space also allows for storage of the output of our Javascript listener, which can get as large as 4 GB per 10,000 sites visited.

Javascript Listening

Originally a daunting task, snooping on the script interpreter in Internet Explorer became much easier with some help from Symantec. We were kindly provided a method for implanting code into Symantec's own script blocker, Canary; Canary calls our code every time a function is called within a script running in Internet Explorer. The relevant parameters that are passed into the code are the Global Unique Identifier (GUID), the Dispatch ID (DISPID, and the IDispatch interface pointer.

The GUID parameter holds the data about the object that is performing the function call. The GUID is stored as a large hexademical value that can be looked up in the Windows Registry to find the specific Internet Explorer component that is being referred to. Each GUID associated with script function calls (around 200 were found) has associated functions that it calls. Each function is, in turn, identified by a DISPID, an integer representing the specific function that the object (represented by the GUID) is calling. The DISPIDs are constant for each object, but values may be similar across different GUIDs. For example, the DISPID 1033 could mean write() with one object, while it could mean open() with another. Due to this, a function's statistical distribution has to be analyzed by first referencing the GUID and then narrowing it down by the DISPID.

The data regarding each function's signature is stored in the IDispatch interface pointer. We extract the function's argument types and values by accessing the getTypeInfo function [3] [4]. The function returns an ITypeInfo pointer, which holds a pointer to the TYPEATTR attribute table, using the GetTypeAttr function [5]. This table contains all of the data necessary to output the function signature, specifically the argument types and even the values. Our code outputs these, along with the GUID and DISPID, for every function call made in the script.

We reviewed the most common argument types, and it was found that DISPATCH, I4 (4 byte Integer), BOOL, BSTR (a string), and NULL make up close to 97% of all arguments. We have thus limited our research to analyzing these argument types. For DISPATCH, our code outputs the other GUID that the function is interacting with. For I4, BOOL, and NULL, our code outputs the value of the variable (NULL holds nothing). For BSTR, our code originally wrote out the string that was in the value, but that was found to be hard to analyze statistically, so instead the size of the string is written.

As an example, consider this simple Javascript function call:

```
window.document.write("THIS IS A TEST STRING OF LENGTH 35.")
```

Running a webpage that contains this script while the callback DLL is loaded yields the following output:

DISPID	GUID	Params	Type 1	Value 1
1151	3050f55d-98b5-11cf-bb82-00aa00bdce0b	0	n/a	n/a
1054	3050f55f-98b5-11cf-bb82-00aa00bdce0b	1	BSTR	35

The first GUID corresponds to the DispHTMLWindow2 object, and its function with DISPID 1151 is called to access its DispHTMLDocument object. The second GUID

is the DispHTMLDocument object grabbed earlier, and in the case of this object, DISPID 1054 corresponds to its *write* function. The function takes one argument, a byte string of length 35.

Python Scripts

A Python script is what ties together the Javascript listening component with the URL list from Heritrix's web crawl. The script has several functions, which are selected from the command line, that include parsing the URL list and visiting each site with Internet Explorer.

In the log file created by Heritrix, there are several fields that are populated that are not necessary for our purposes. A sample line looks as follows:

```
2004-07-21T23:29:40.438Z       200     310
http://127.0.0.1:9999/selftest/Charset/charsetselftest_end.html LLLL
http://127.0.0.1:9999/selftest/Charset/shiftjis.jsp text/html #000
20040721232940401+10 M77KNTBZH2IU6V2SIG5EEG45EJICNQNM -
```

The actual URL of each site is recorded in the sixth field, which in this sample line is /selftest/charset/shiftjis.jsp. The Python script extracts this field and writes it into a new, smaller log file.

The script then runs through the parsed log file and opens each URL in Internet Explorer, which is being monitored by our Javascript listening component. We access Internet Explorer reliably using the PAMIE [6] library. There were initial issues with the timing of opening and closing Internet Explorer because scripts take a longer time to load as they are going through the Javascript interpreter. To accommodate this, our Python script waits for 100 seconds before forcibly closing Internet Explorer if it does not exit right away.

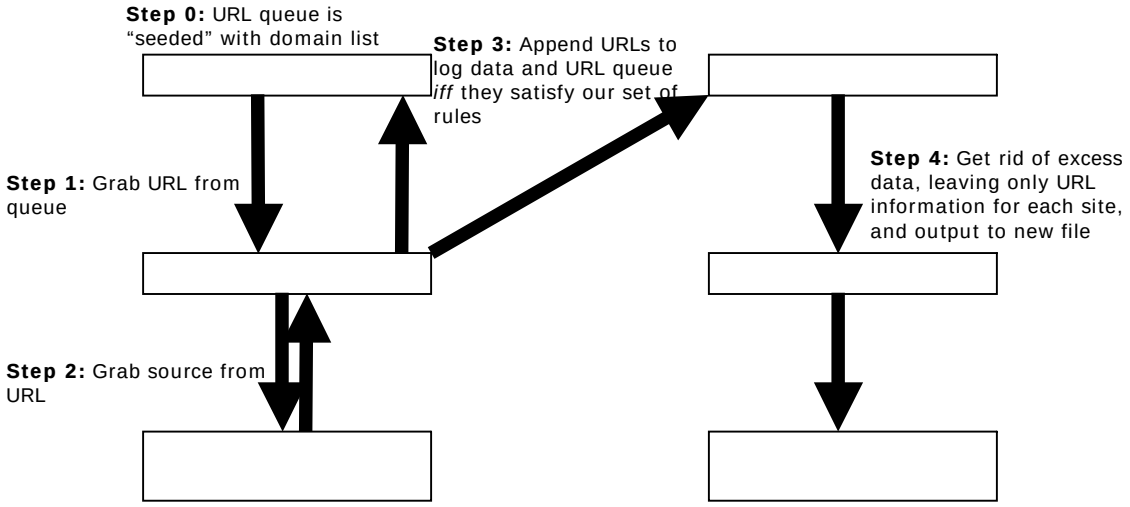
Python scripts are also being used to analyze the output of the Javascript listener. Because the files generated are over 4 GB, a standard database cannot be used to make queries and find statistical trends. To solve this issue, a script was written to read the file line by line and create a statistical analysis of the data, such as the smallest point of data, the largest point, the average, the standard deviation, and so on.

Flowcharts

Several components had to be integrated together to create a functional infrastructure, and the tasks of finding or creating the components and tying them together to create our system took about a full quarter, with another quarter mainly dedicated to data gathering over the web.

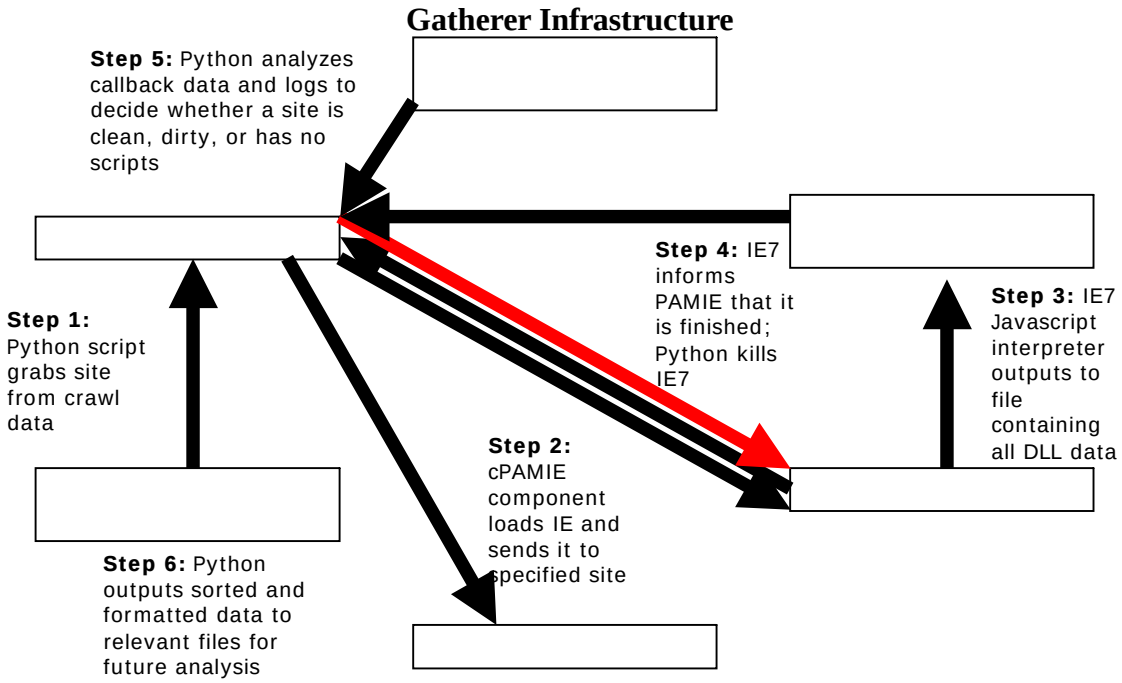
The following flowcharts provide an overall view of how our built infrastructure functions. The crawler is used to quickly grab as many links as possible and pool them together, while the gatherer is used to process each URL using our CanaryCallback DLL.

Crawler Infrastructure



Repeat steps 1-4 until crawl limit is reached.

Gatherer Infrastructure



Repeat steps 1-6 until URL list is exhausted.

Research: Initial Approaches with Focus on Integer Analysis

After getting our first data sets, we started to come up with preliminary ways of analyzing the sets as a whole. We gathered basic information such as the most commonly used objects and functions. Next, we looked at specific functions which used integer arguments. We chose functions that took integer arguments because integers would be the easiest type of argument on which to perform basic analysis. We ran these same object commonality and function argument distribution tests on 3 separate runs, documenting the results. We then looked at the integer argument functions of the third data set with several more different approaches, hoping to find patterns allowing us to characterize normal behavior.

Result Set 1 – Common Functions and 3 Specific Functions

In the first run, we crawled 5 seed websites: Google, Yahoo, MSN, Youtube, and MySpace. We ran our logging script on these sites, but our script crashed after visiting 1324 web pages. These pages gave us 3,469,151 function calls, for an average of about 2620 function calls per page. Our analysis of the data follows:

Result Set 1 - Most Common Objects:

	Object	Count	% of Total
1	DispHTMLWindow2	802784	23.14
2	DispDOMChildrenCollection	614898	17.72
3	DispHTMLDivElement	433562	12.50
4	DispHTMLAnchorElement	175769	5.07
5	DispHTMLElementCollection	151554	4.37

There were 108 different GUIDs. The most common one made up more than 23% of all the GUIDs in our function calls. It is notable that more than half of all the function calls originate from just 3 distinct GUIDs. We can look up GUIDs in the Windows Registry to find the objects to which they correspond.

Result Set 1 - Most Common Functions:

	Object	DispID	Count	% of Total
1	DispDOMChildrenCollection	1500	382832	11.04
2	DispHTMLDivElement	-2147417063	132307	3.81
3	DispDOMChildrenCollection	1000000	127752	3.68
4	DispHTMLWindow2	1151	108663	3.13
5	DispHTMLWindow2	3000008	101838	2.94

A function is uniquely identified with the combination of its GUID and DispID. There were 7,361 unique functions. The most common are listed above. The most common function makes up 11% of our function calls. The frequency of the second most common drops off steeply, making up less than 4%. The most common function is a call by the DispDOMChildrenCollection object. Another call by this object is also among the most common functions, in position 3.

Result Set 1 - Most Common Int-Argument Functions:

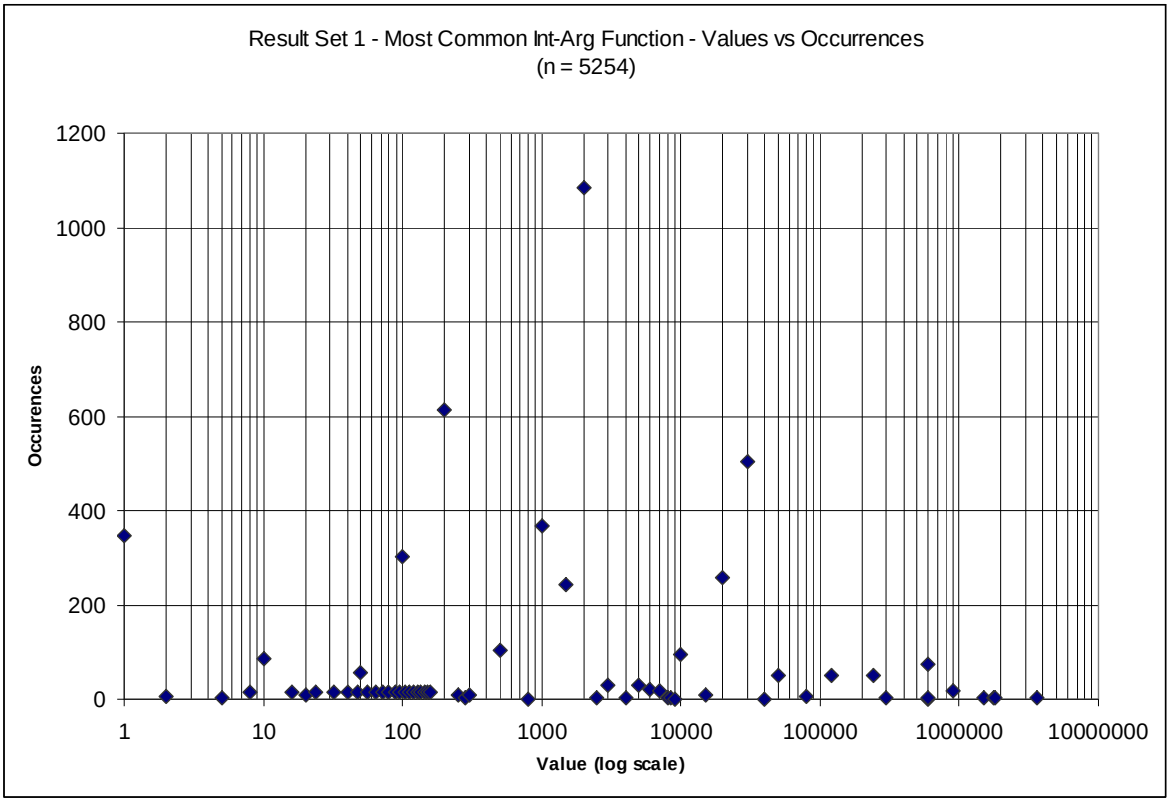
	Object	DispID	Count
1	DispHTMLWindow2	1103	5254
2	DispHTMLDocument	1013	1510
3	DispHTMLIFrame	-2147418107	1506
4	DispHTMLIFrame	-2147418106	1504
5	DispHTMLWindow2	1104	1049

The vast majority of function calls take no arguments. These functions are hard for us to analyze. Functions with integer arguments are the easiest for us to analyze, so we start by looking at the most common functions that take in integer arguments. 932 out of our 7,361 functions take an int argument. This subset of functions are rather uncommon compared to the most common overall functions; the most common int-argument function has 5,254 occurrences, while the most common overall function has 382,832 occurrences. It is worth noting that with over five thousand occurrences, the most common int-argument function is much more common than any of the other int-argument functions. The next three most popular functions all have a similar number of occurrences, at slightly over 1,500 each.

Result Set 1 - Most Common Int-Argument Functions in Detail

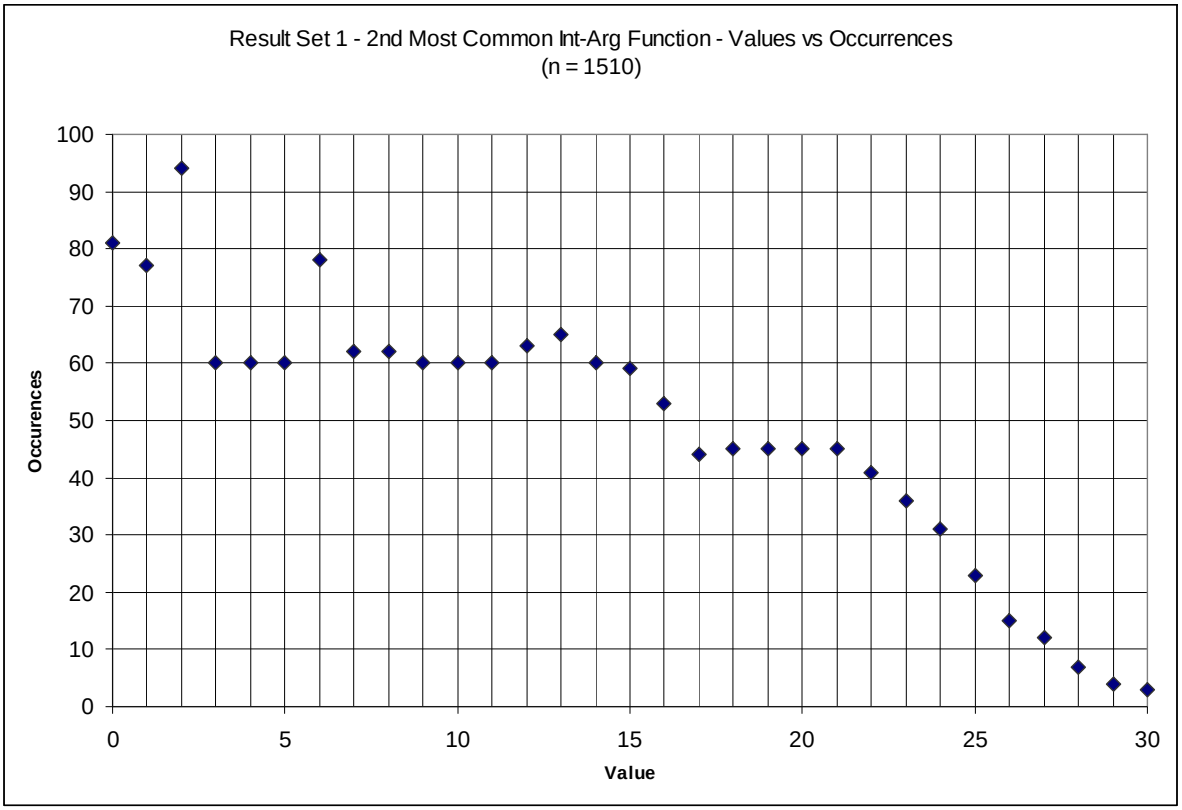
Next, we analyze the most popular int-argument functions in detail. We take a look at the arguments of the top 3 functions in an attempt to find a pattern of regular behavior for the functions with regards to arguments.

Result Set 1
Most common int-arg function:
GUID: 3050f55d-98b5-11cf-bb82-00aa00bdce0b (DispHTMLWindow2)
DispID: 1103
Occurrences: 5254



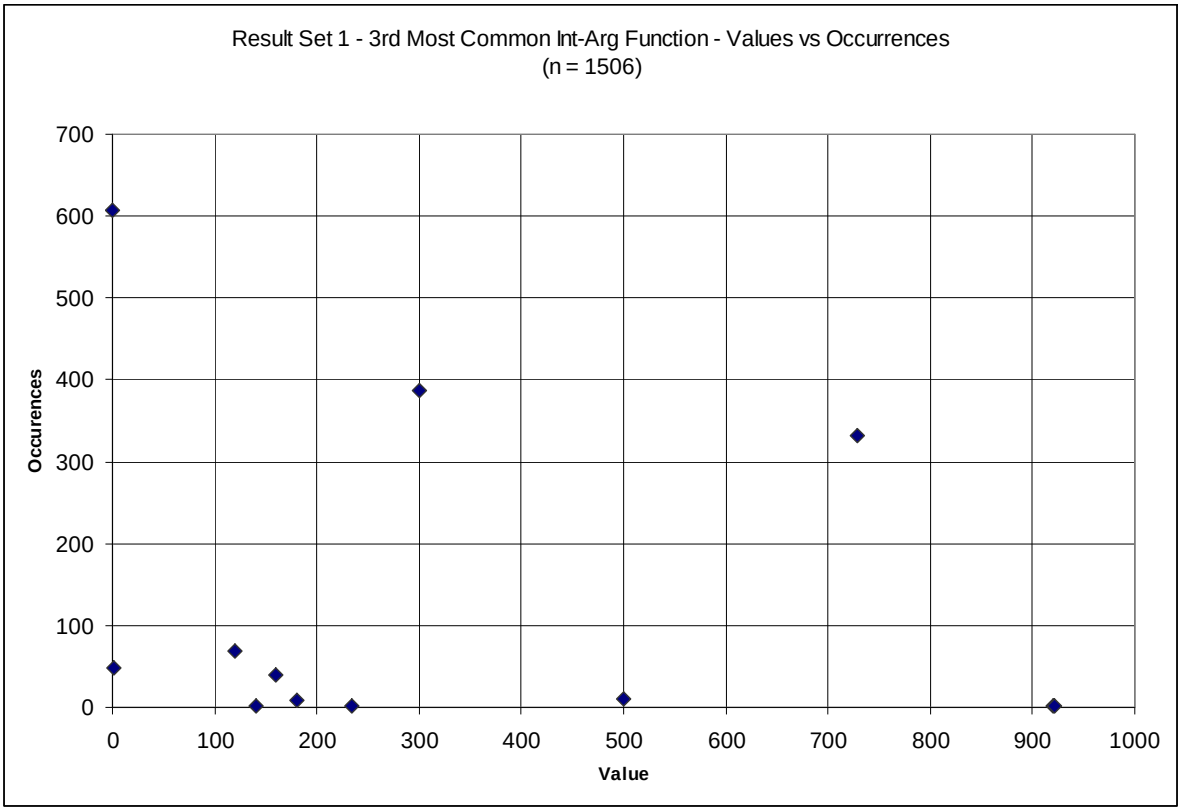
The function's arguments have a very large range, from a low of 0 to a high of 3.6 million. The most popular values are 2000, 200, and 30000, with 1085, 613, and 504 occurrences respectively. The distribution of the number of occurrences doesn't seem to follow any pattern. However, the 5254 arguments are each one of 64 distinct values. The fact that all the arguments to this function seem to be one of a relatively small set of values suggests that there is some normalcy in the behavior of the graph.

Result Set 1
Second most common int-arg function:
GUID: 3050f55f-98b5-11cf-bb82-00aa00bdce0b (DispHTMLDocument)
DispID: 1013
Occurrences: 1510



This function has a more regular distribution for argument values. The range of the values is from 0 to 30. Each integer within the range has occurrences. Lower values are the most common, with an average of about 65 occurrences for the numbers 0-16. As the argument values get higher they occur less frequently, with a rapid drop-off starting with value 22.

Result Set 1
Third most common int-arg function:
GUID: 3050f51b-98b5-11cf-bb82-00aa00bdce0b (DispHTMLIFrame)
DispID: -2147418107
Occurrences: 1506



From this graph, the values in this function don't seem to have a particular pattern. The range of values is from 0 to 921. There are only 12 distinct values used as arguments for this function. By far the most common 3 values are 0, 300, and 728, with 607, 387, and 332 occurrences, respectively. The other 9 values occur less than 70 times each, with 4 of them occurring only once.

Result Set 2 – Common Functions and 3 Specific Functions

After we looked at our first data set, we decided to run our scripts on another, different set of data in order to test the robustness of our first data set. In our next run, we crawled 10 seed websites: Google, Yahoo, MSN, Youtube, MySpace, Facebook, Live.com, Wikipedia, Ebay, and Amazon. Although this script also crashed before completion, it ran through 1184 web pages, giving us significant data. These pages gave us 3,706,454 function calls, for an average of about 3130 function calls per page. This number is significantly higher than the average number of calls per page for our first run. Our analysis of the data follows:

Result Set 2 - Most Common Object:

	Object	Count	% of Total
1	DispHTMLWindow2	1010109	27.25
2	DispHTMLDivElement	353158	9.53
3	DispHTMLDocument	350513	9.46
4	DispHTMLElementCollection	326865	8.82
5	DispHTMLAnchorElement	298534	8.05

This time there were only 95 different GUIDs. The most common one made up more than 27% of all the GUIDs in this set of function calls. It is by far the most common GUID in this result set. The first, second, and third most common GUIDs from the first result set are now the first, twentieth, and second most common GUIDs from this result set. This result set shares 4 out of 5 of the most common GUIDs with our previous result set, but the second most common object in the first result set, DispDOMChildrenCollection, is now much less common.

Result Set 2 - Most Common Functions:

	Object	DispID	Count	% of Total
1	DispHTMLWindow2	1151	289132	7.80
2	DispHTMLAttributeCollection	1500	141100	3.81
3	DispHTMLDOMAttribute	1001	139470	3.76
4	DispHTMLDocument	1030	95754	2.58
5	DispHTMLWindow2	10002	72471	1.96

There were 5,577 unique functions in this result set, less than the amount in the first result set. The most common function makes up 7.8% of our function calls. The frequency of the second most common drops off steeply, making up less than 4%. Note that the most common five functions in the first result set and second result set share only 1 function between them: DispHTMLWindow2 is ranked 1 and 5 in this set, and ranked 4 and 5 in the first result set. The other three objects in this result set are not among the five most common functions of the first result set. Also note that while the DispHTMLWindow2 object is among the most common functions in both of our data sets, the specific function determined by its Dispatch ID is different in each of the result sets.

Result Set 2 - Most Common Int-Argument Functions:

	Object	DispID	Count
1	DispHTMLWindow2	1103	4406
2	DispHTMLDocument	1013	4112
3	DispHTMLWindow2	1162	2057
4	DispHTMLWindow2	1163	1813
5	IHTMLImageElementFactory	0	1794

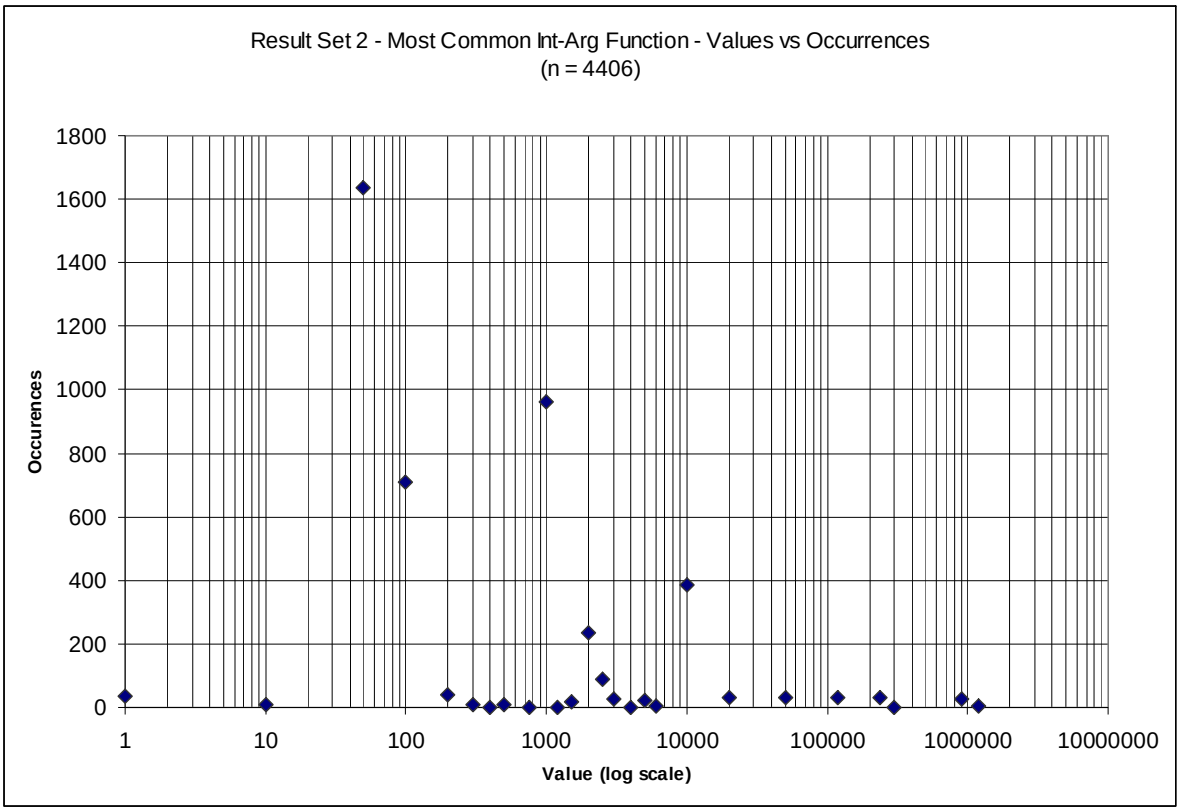
Like our first result set, the vast majority of function calls take no arguments. This set has only 490 int-argument functions out of its 5,577 unique functions. The top two most common int-argument functions of this result set are the same as in the first result set, but the next three most common of this set are not in the top five most common of the first set. The third most common function of the first result set is actually the 95th most common here, which is a significant difference between the two data sets.

Result Set 2 - Three Int-Argument Functions in Detail

In order to check for robustness and consistency, we analyzed the same three int-argument functions that we analyzed in the last dataset.

Result Set 2

Most common int-arg function (also most common in previous result set):
GUID: 3050f55d-98b5-11cf-bb82-00aa00bdce0b (DispHTMLWindow2)
DispID: 1103
Occurrences: 4406

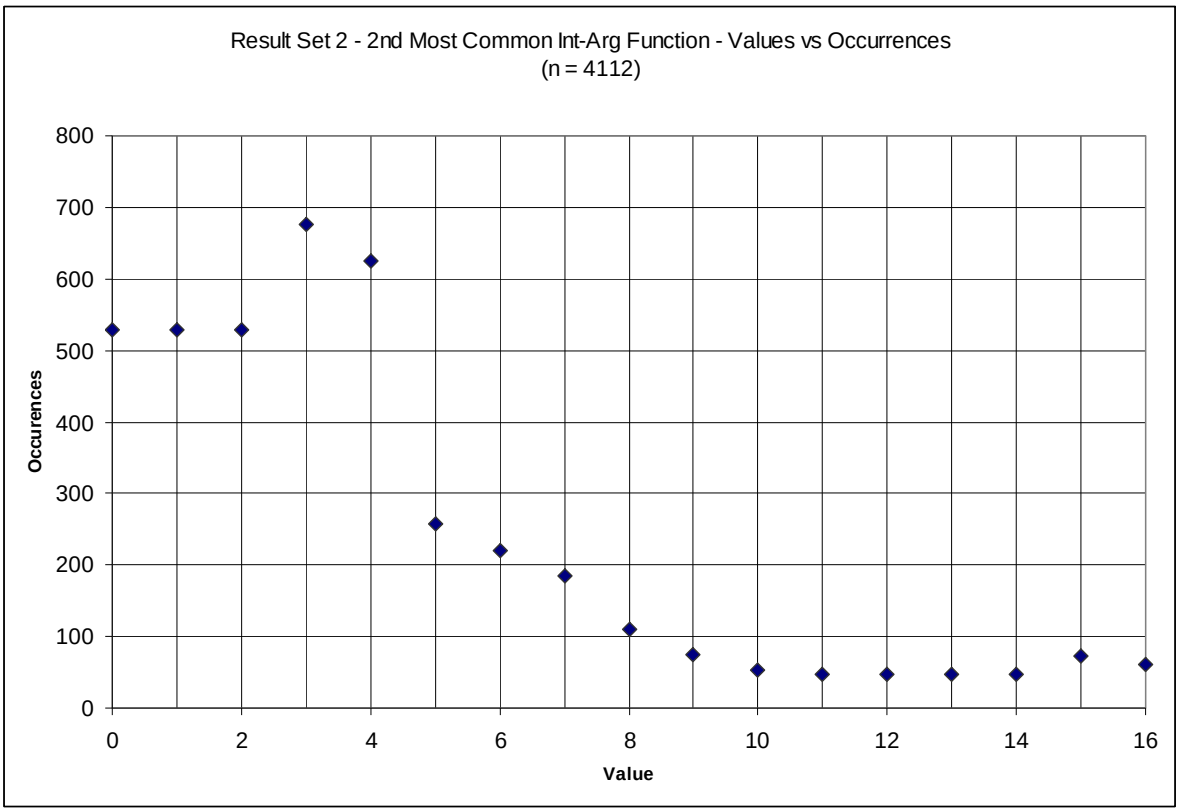


Like in the first set, the function's arguments have a very large range. But this time the range is smaller by a factor of 3, from 0 to 1.2 million. The three most popular

values are 50, 1000, and 100, with 1634, 961, and 708 occurrences respectively. These values are not the same as the most popular values of this function in the first set. The first set's most popular value, 2000, is this set's fifth most popular value. In the first set, the values were one of 64 distinct values. In this set, there are only 27 distinct values. These are several big differences that this function has between the first set and this set.

Result Set 2

Second most common int-arg function (also second most common in the first result set):
GUID: 3050f55f-98b5-11cf-bb82-00aa00bdce0b (DispHTMLDocument)
DispID: 1013
Occurrences: 4112



In the second result set, this function has a similar distribution to what it had in the first result set. Lower values are the most common, and as the argument values get higher their number of occurrences slowly drops off. However, a significant difference this time is that the range of the values is from 0 to 16. In the first result set, it was from 0 to 30.

Result Set 2
95th most common int-arg function:
GUID: 3050f51b-98b5-11cf-bb82-00aa00bdce0b (DispHTMLIFrame)
DispID: -2147418107
Occurrences: 21

<No Graph>

A big change between the first and second result sets is the prominence of this function. In the first result set, this function is the third most popular int-argument function. In the second result set it is the 95th most popular. In the first data set, the function took in a variety of arguments, ranging from 0 to 921. In this data set, all 21 calls to this function take the number 0 as their arguments.

Conclusions Regarding First Two Sets

From analyzing these three functions in these first two data sets, we have gotten a great deal of information regarding our project. We have learned that some functions do tend to have some consistency between sets of data. The first functions had a relatively small set of possible values from a large range and a relatively random distribution for both of our data sets. The second functions both had a common sloping pattern. Seeing results like this tells us that there may be a way to get common characteristics from each function to assist us in our goal of testing for "normal" behavior.

However, there were many inconsistencies between our data sets. First, the list of the most common functions and most common int-functions has significant differences between the two data sets. Second, the values that the three tested functions take vary significantly between the two data sets. Some functions, like the third one we looked at, seem like they have nothing in common between the data sets. Its occurrence rank was three in one set and 95 in the other, and it took in a variety of arguments in the first data set, but only one argument in the second set.

We theorize that some of these inconsistencies are a result of selection bias in our data collection. We gathered our data using a rather small set of seed sites. The first set had 5 seeds, and the second set had 10 seeds. We think that this is much too small of a sample set to detect consistent "normal" behavior of functions. If some of the seed sites call a particular function many times as part of a common script within its domain, it may significantly influence our dataset. Functions or arguments that are typically rare may become very common in our data as a result of being common in one of our seed sites. To remedy this, our next run will have 200 seeds. This will give us a much broader range of function calls, instead of constantly getting the same few functions from the same few domains.

Ideas for analyzing future data include function ordering and conditional patterns. So far we consider each function independently. We do not consider the function's context: what other functions occur on the same page, and the order in which functions on a page are called. If certain functions are regularly preceded by or followed by certain other functions, this information will be helpful in our goal of characterizing function behavior. Conditional patterns can be another next step for us to explore. Our results show indications of selection bias, in which specific functions are called with different

frequencies and different arguments depending on the site we are observing. An idea to take advantage of this is to group our sites into separate categories with each group having similar usage patterns for these specific functions. For example, a certain group of sites may call a function with only arguments X, while another group calls that function with only arguments Y. When we analyze a new site, we can then analyze its use of that function and determine whether it matches a known group.

Result set 3 - Common functions and 3 specific functions

Our third result set, with 200 seeds, analyzes a much larger set of data. The script went through the entire crawl of 15,796 web sites, over 10 times more sites than in the previous two result sets. These sites provided us with 88,698,817 function crawls, over 20 times more than the previous two sets. There was an average of 5,615 functions per site. This was significantly more than in our previous result sets.

Like in our previous sets, we started by analyzing the functions with integer arguments. About 0.84% of the 88 million function calls had integer arguments. This provided us with a sample size of 743,270 integer argument functions to look at. There were 4,269 distinct integer argument functions, much more than in the previous two sets.

Result Set 3 - Most Common Int-Argument Functions:

	Object	DispID	Count
1	DispHTMLElementCollection	0	57810
2	DispHTMLWindow2	1103	53874
3	DispHTMLWindow2	10002	12799
4	IHTMLImageElementFactory	0	11925
5	DispHTMLDocument	1013	11418

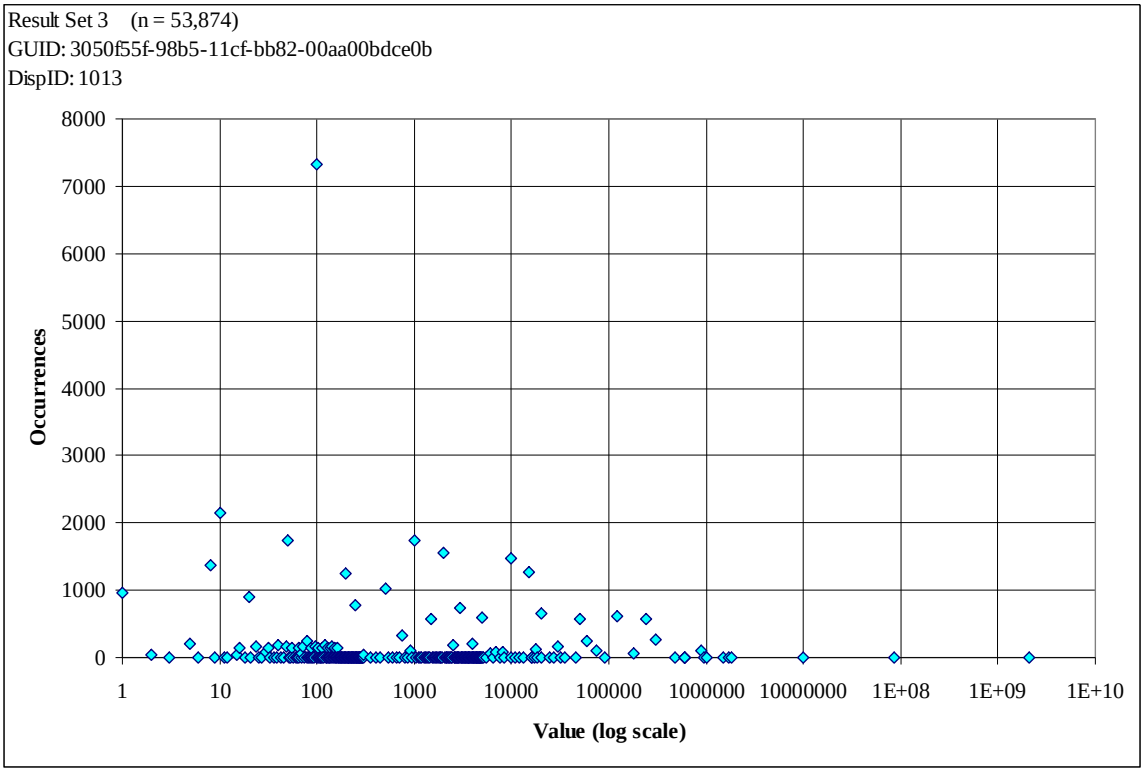
The results of our third data set has both similarities and differences with our previous two data sets. The DispHTMLElementCollection object was one of the 5 most common objects in both of the two previous data sets, although it was not among the most common int-argument functions. Interestingly in this set, one of its function calls is the most called int-argument function. The other three objects in this top 5 list are all among the top 5 int-argument functions of the previous two result sets.

Result Set 3 - Three Int-Argument Functions in Detail

Checking for consistency, we analyzed the same three int-argument functions that we analyzed in the previous datasets.

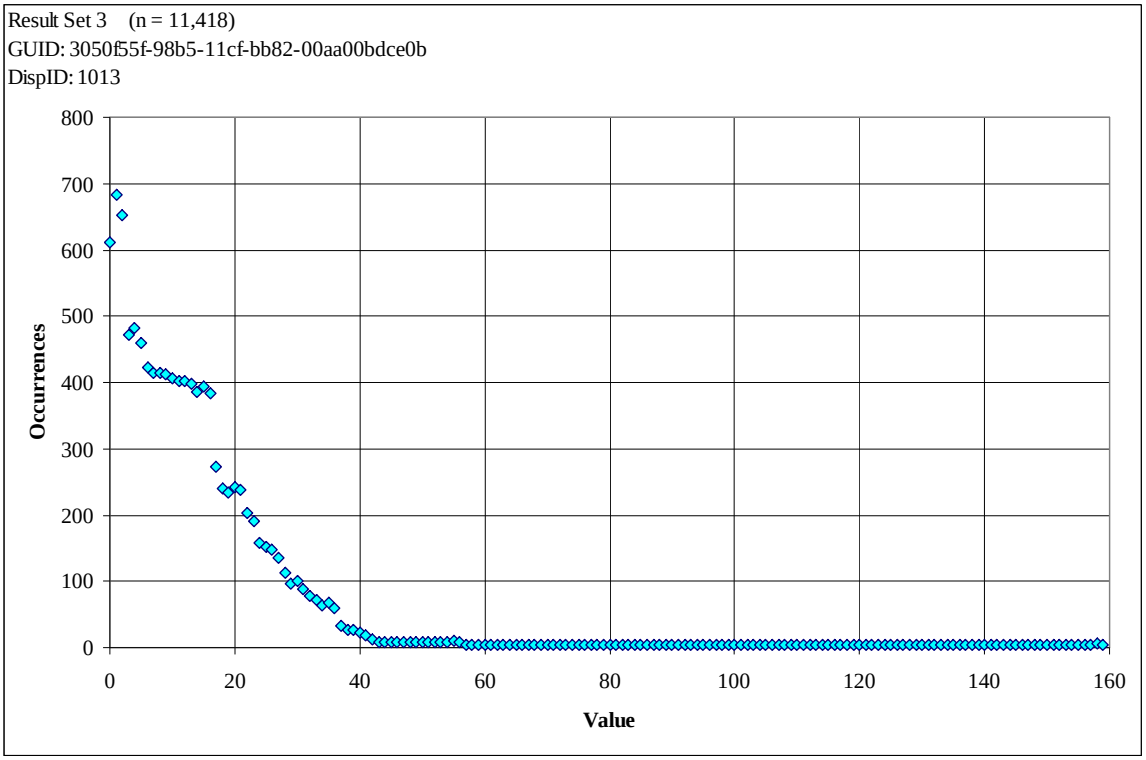
Result Set 3

Second most common int-arg function (most common in previous result sets):
GUID: 3050f55d-98b5-11cf-bb82-00aa00bdce0b (DispHTMLWindow2)
DispID: 1103
Occurrences: 53874



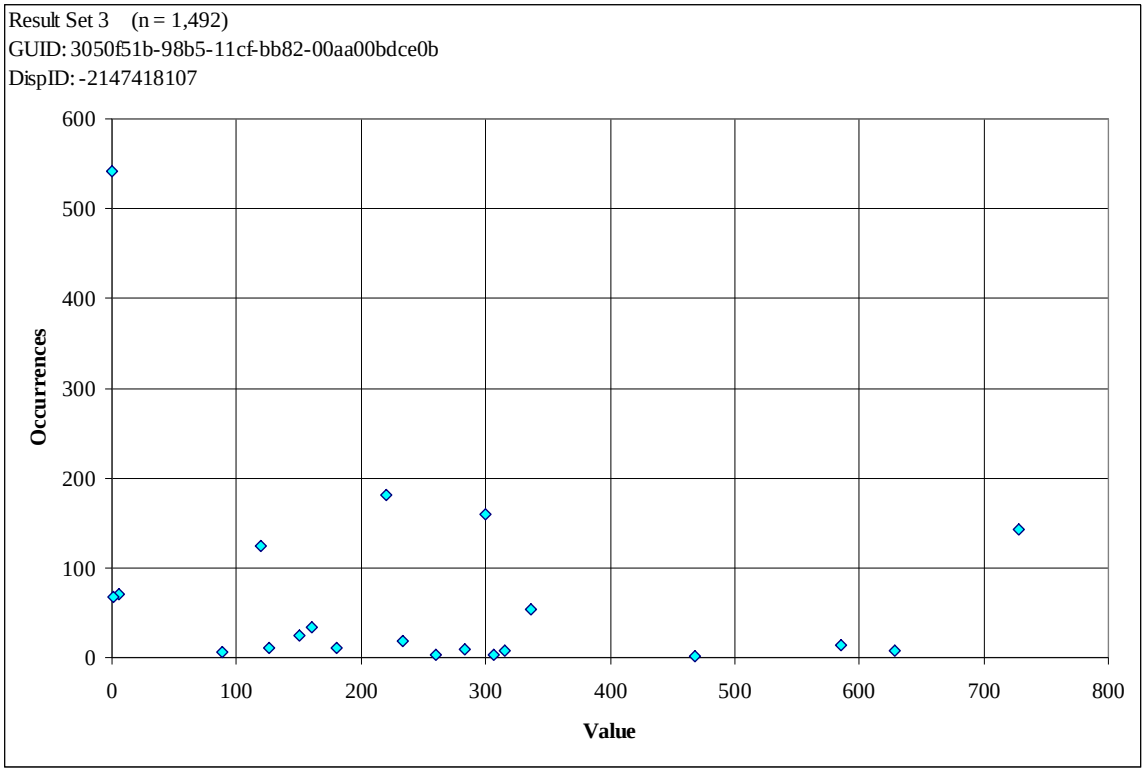
As we have seen in this function previously, it has a very large range. The range this time goes from 0 to 2.07 billion. This is much larger than before, although the largest values are outliers with very few occurrences. 900,000 seems to be the largest argument that occurs relatively commonly, with 109 occurrences. The 8 arguments larger than 900,000 each occur 1 to 4 times. The most common values this time were 0, 100, and 10, with about 19 thousand, 7 thousand, and 2 thousand occurrences respectively. Note that in this chart, the most popular value of 0 is cut off. The value 100 was the third most common value in the second result set, but 0 and 10 were not among the most common in the previous two sets. These values again confirm that the behavior of this function is somewhat erratic. This time, the number of distinct values that the function took was 270. This is larger than the previous two sets, which had 64 and 27 distinct values. This can be explained because the sample size this time is 10 times as large as it was previously.

Result Set 3
Fifth most common int-arg function (second most common in the previous result sets):
GUID: 3050f55f-98b5-11cf-bb82-00aa00bdce0b (DispHTMLDocument)
DispID: 1013
Occurrences: 11418



This function has had consistent behavior among all 3 result sets. In each set, the most common arguments are smaller integers. As the arguments get larger, they slowly taper off. This third result set has a much longer tail than the previous two, extending all the way to 159. Each of the integers from 57 to 157 has exactly 4 occurrences. 158 and 159 have 6 and 5 occurrences, respectively.

Result Set 3
31st most common (3rd most common in first run, 95th most common in second run)
GUID: 3050f51b-98b5-11cf-bb82-00aa00bdce0b (DispHTMLIFrame)
DispID: -2147418107
Occurrences: 1492



The behavior of this function matches with its behavior in the first result set. The four most common values this time are 0, 220, 300, and 728, with occurrences of 542, 181, 160, and 143 respectively. This somewhat matches up with the most common values in the first result set: 0, 300, and 728. Unlike in this set, 220 did not occur in the first set, however if 0, 300, and 728 are in fact always common among this function then it will be a possible way to categorize normalcy for this function. A caveat is that this function is not among the most common int-argument functions. In the first result set it was third most common, which made it common enough to analyze. In this result set it was 31st most common (out of 4269), which combined with the large sample size of data, gave enough data for us to analyze it. However, in the second run this function was 95th most common, and only had 21 occurrences.

Result Set 3 - Dispatch IDs in Integer Argument Functions

We wanted to look at the problem of characterizing normal behavior from many different angles, so we took a broader approach next. We analyzed the occurrences of each dispatch ID of all the integer argument functions, without taking GUID into consideration.

There were 1,242 distinct dispatch IDs used in integer argument functions. These ranged from -2,147,418,109 to 3,001,286. We discovered that almost all of the function

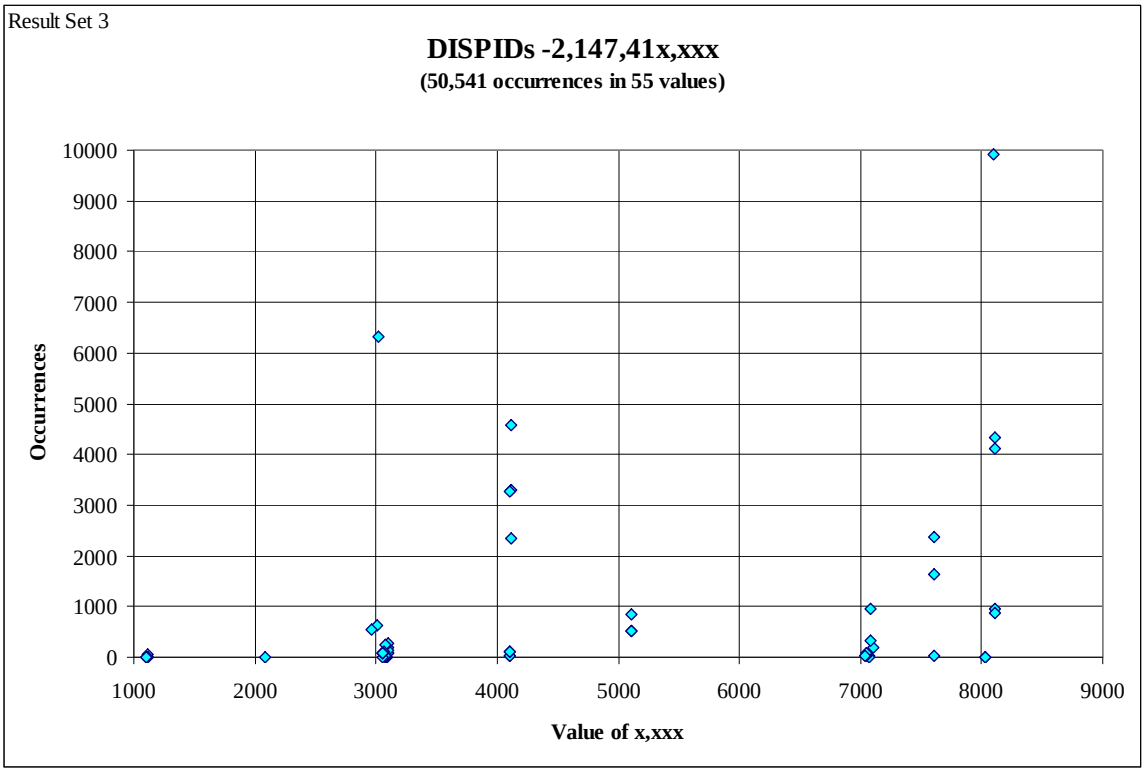
calls were grouped into four different subranges within this large range, with one subrange having the majority. Of the dispatch IDs of the 743,270 function calls:

- 50,541 are one of 55 numbers between -2,147,418,109 and -2,147,411,105
- 164,224 are one of 39 numbers between 0 and 2,313
- 38,099 are one of 75 numbers between 10,001 and 10,087
- 490,201 are one of 1,067 numbers between 3,000,000 and 3,001,286

205 of the functions were exceptions to these four groups. These exceptions all had dispatch IDs of 5425, 7001, 7002, 27001, 1000001, or 1333371.

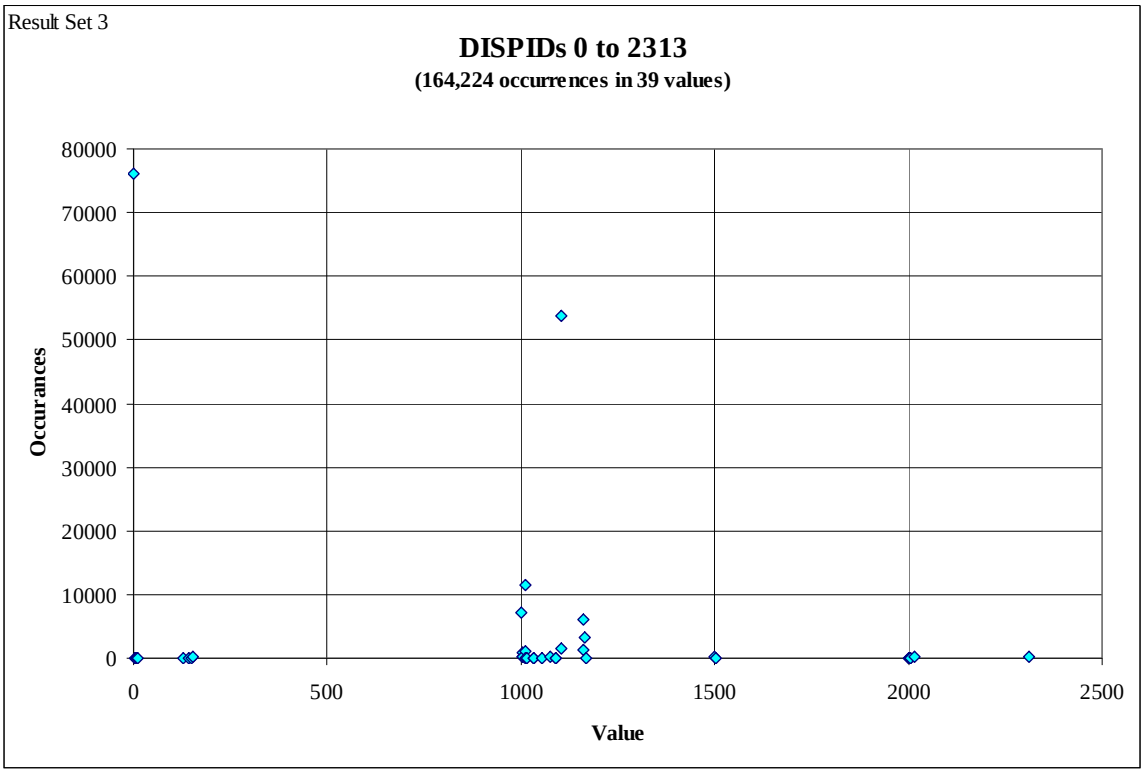
Next, we took a closer look at each subrange, looking at the occurrences of the dispatch IDs within each one.

First Dispatch ID Subrange: -2,147,418,109 to -2,147,411,105



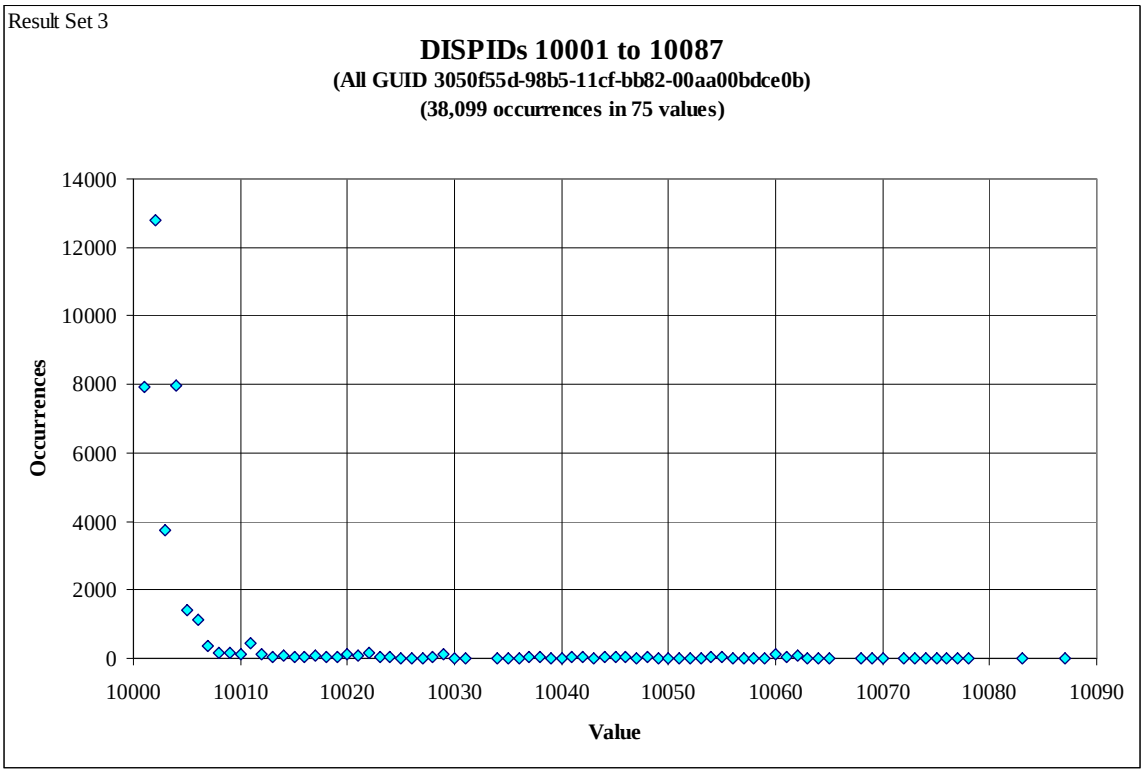
50,541, or about 6.8% of the functions, fall in this range. There are 55 distinct dispatch IDs used in this range. Note that the number 2^{31} is 2,147,483,648, which is very close to our range. This may indicate an improper conversion from an unsigned long into a signed long in our scripts. The two most common functions in this range were -2,147,418,097 and -2,147,413,021, with 9,907 and 6,308 occurrences respectively. The majority of the dispatch IDs used are grouped around the round numbers ending in thousands. Although no dispatch ID occurred in exactly a round thousand, almost all of them are within 120 below each round thousand (ending in 001-120). The only exceptions are the four dispatch IDs ending in 2959, 7605, 7610, and 7611.

Second Dispatch ID Subrange: 0 to 2,313



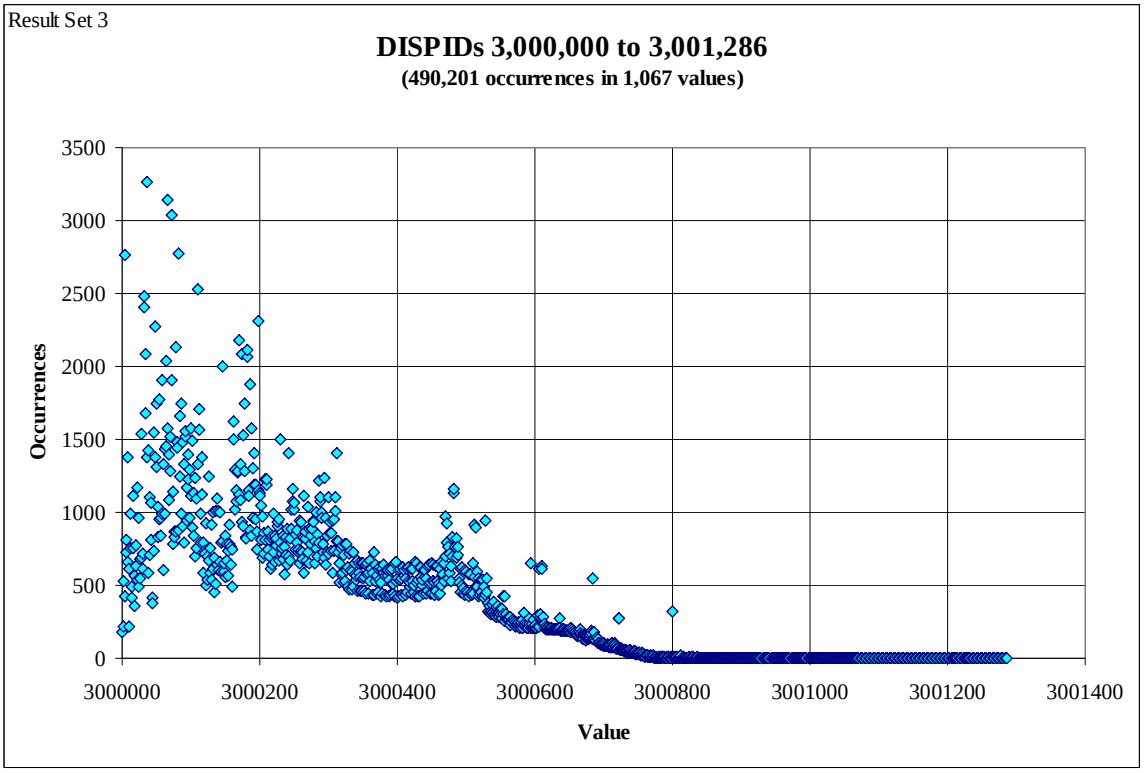
164,224, or about 22.1% of the functions, fall within this range. There are 39 distinct dispatch IDs used in this range. By far, the two most common are 0 and 1103, with 76,070 and 53,874 occurrences respectively. Within this subrange there appear to be groups of dispatch IDs in several sub-subranges grouped around certain numbers. The dispatch IDs used can be split into subgroups of the numbers: 0-9, 127-154, 1002-1168, 1500-1504, 2001-2015, and the number 2313.

Third Dispatch ID Subrange: 10,001 to 10,087



38,099, or about 5.13% of the functions, fall within this range. There are 75 distinct dispatch IDs used in this range, so the majority of the integers in the range are used. This subrange is unique in that the entire range comes from one particular object: the DispHTMLWindow2 object, with GUID 3050f55d-98b5-11cf-bb82-00aa00bdce0b. dispatch IDs with lower numbers are the most common, tapering off quickly after 10007. 10002 is the most common, with 12,799 occurrences. Other dispatch IDs between 10001 and 10006 have between 1,124 and 7,971 occurrences, and all IDs above 10006 have occurrences in the triple digits or less.

Fourth Dispatch ID Subrange: 3000000 to 3001286



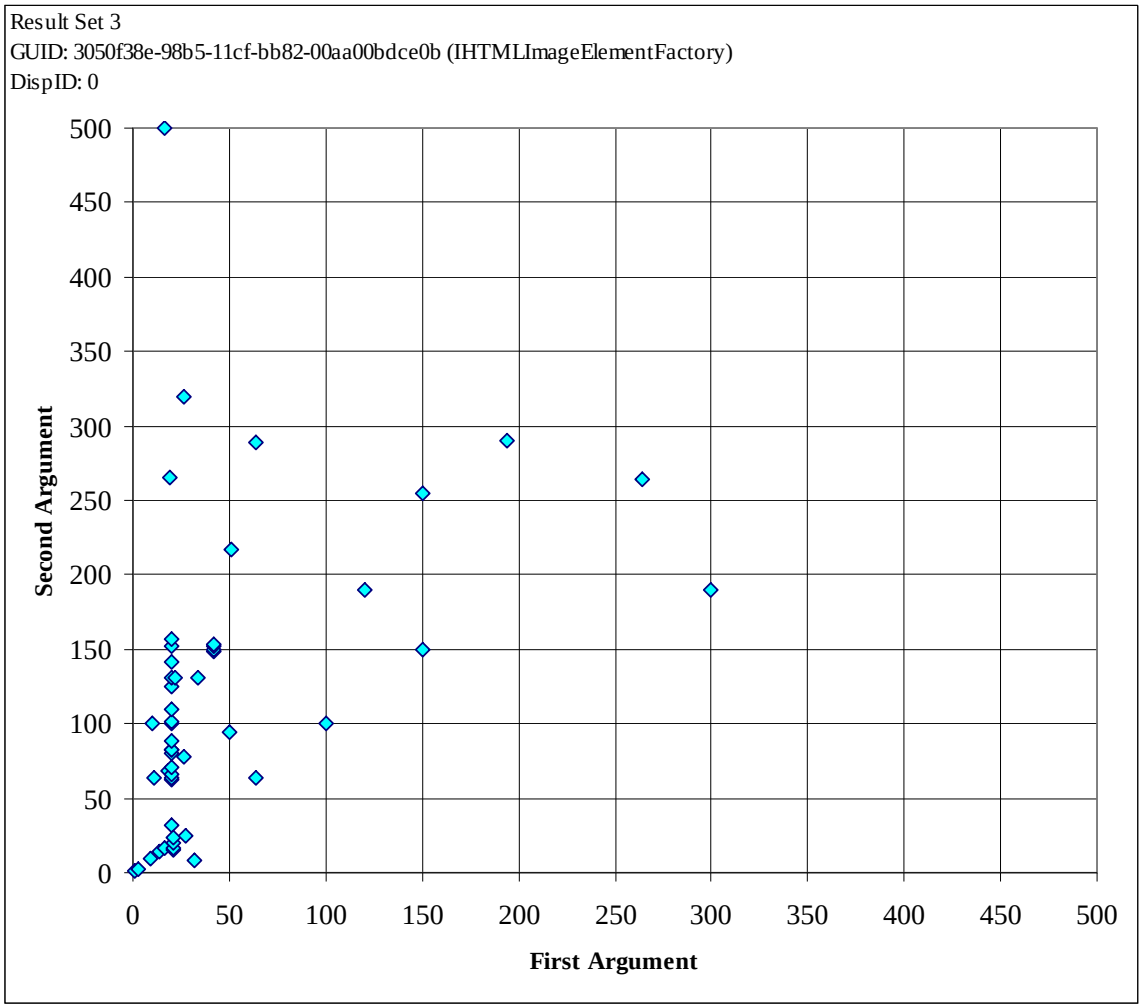
490,201, or about 65.95% of the functions, fall within this range. There are 1,067 distinct dispatch IDs used in this range, so the majority of the integers in the range are used. Similar to the third dispatch ID subrange, the lower dispatch IDs in this range tend to be more common, tapering off as the values get larger. This subrange has much more variance within the lower numbers, but the tapering off can clearly be seen:

- The first 200 dispatch IDs, 3000000-30000199, have 1,121 occurrences each on average.
- The second 200 IDs have 737 occurrences on average.
- The third 200 IDs have 471 occurrences on average.
- The fourth 200 IDs have 121 occurrences on average.
- All the IDs above 3000799 have occurrences of 15 or less, with the majority having only 1 occurrence.

Multiple Integer Arguments in Integer Argument Functions

Another approach that we used to look for patterns was to look at functions with more than one integer argument. We wanted to see if there was any pattern between the two (or more) integer arguments of the function. These functions turned out to be relatively rare. We found several functions with two integer arguments.

The most popular function with multiple integer arguments was the IHTMLImageElementFactory object, with dispatch ID 0. This had 11,925 occurrences. This function took two integers as arguments. Almost 95% of the time this function was called with the arguments (1,1) or (3,2). The other 5% of the time, it was called with other arguments, seen in the graph below:



We could not find any regular relation between the two arguments in the function. Sometimes the two arguments were the same; other they were different. Sometimes the first argument was larger than the second; other times it was smaller. Some values of one argument were called only with a particular value for the other argument; values of some arguments had multiple corresponding values for the other argument. The arguments weren't multiples of each other, and didn't seem to follow any other patterns.

Other functions that we looked at also failed to produce any significant results. The function with GUID 3050f55d-98b5-11cf-bb82-00aa00bdce0b (DispHTMLWindow2) and dispatch ID 1160 had two integer arguments, but the first one

was always 0. The function with GUID 3050f38c-98b5-11cf-bb82-00aa00bdce0b (IHTMLOptionElementFactory) and dispatch ID 0 had two integer arguments. However, there were only 326 calls to this function, and they all came from one of two pages: http://signup.myspace.com/index.cfm?fuseaction=join and http://signup.myspace.com/index.cfm?fuseaction=join&MyToken=d233f564-246c-49d6-9731-b71df1856. The two arguments in this function were always the same. The each symmetrical argument from (1,1) to (31,31) occurred twice, and each symmetrical argument from (1908,1908) to (2008,2008) occurred once. The arguments of these functions seem to correspond to the days and years of dates, which is supported by the fact that the two web pages which call this function are signup pages. The two arguments of the function are definitely correlated in that they are the same, however the small sample size of this function (only on 2 web pages) is much too small to make any conclusions about the normal behavior of this function.

Patterns in Function Order

While looking through the data set of integer functions, we noticed that some functions one particular GUID seemed to often be followed by a certain function, and the pair of functions would have similar arguments. We filtered out these functions to try to find a regular pattern:

We looked at calls to the object 3050f55d-98b5-11cf-bb82-00aa00bdce0b (DispHTMLWindow2) with dispatch IDs ranging from 10001-10062. There were 38,099 of these functions. 3,595, or about 9.4% of these functions were followed by the function with GUID c59c6b12-f6c1-11cf-8835-00a0c911e8b2 and dispatch ID 0. This second function did not appear anywhere else other than the 3,595 occurrences following the DispHTMLWindow2 function. Looking at pairs of these functions, the arguments of the DispHTMLWindow2 function and the c59c6b12-f6c1-11cf-8835-00a0c911e8b2 function were very similar. The DispHTMLWindow2 function had a variety of numbers and types of arguments, ranging from 1 to 9, of types integers, bools, nulls, bstrings, and dispatches. The c59c6b12-f6c1-11cf-8835-00a0c911e8b2 functions always had 1 more argument than the DispHTMLWindow2 functions that it followed. The first argument is always a dispatch, and the rest of its arguments are an exact copy of the DispHTMLWindow2 function's arguments.

This is the case for every occurrence of the c59c6b12-f6c1-11cf-8835-00a0c911e8b2 function in our result set. This leads us to believe that the regular behavior for this function is to always follow DispHTMLWindow2 functions with a certain dispatch ID range. The arguments of the c59c6b12-f6c1-11cf-8835-00a0c911e8b2 function should be a dispatch, followed by the arguments of the previous function. If a c59c6b12-f6c1-11cf-8835-00a0c911e8b2 function is found not following this pattern, it would be an abnormal occurrence that does not fit our result set.

Conclusion with Analysis of Third Data Set

The third result set is a much larger collection of data than our previous two result sets, with a much broader range of seeds. Theoretically it should be a better representation of the internet in general.

We first analyzed integer functions in the same way that we did in the first two result sets. Comparing the previous data against the third set's data gives us evidence for

whether the behavior that we found applies in general over the internet, or whether it is specific to the small sample set of the first two data sets.

We looked at the most common integer argument functions, and found that although our third set had similar most common functions to the other sets, the most popular function of the third set, DispHTMLElementCollection with dispatch ID 0, was not among the most popular of the first two sets. This supports the idea that although this particular function is not used very often within the seed sites of our first two sets, its usage in general on the internet is much higher.

We also looked the behavior of the three particular functions that we analyzed in the two previous sets. The behavior of these functions in the third result set turned out to be quite similar to the behavior of the functions in the previous two sets. The first function's arguments are a small set out of a very large range, but the particular values and amount of occurrences of each argument tend to be erratic. The second function followed a similar pattern in all three result sets: smaller integers are most common, with the occurrences of larger values gradually dropping off. The results of the third function in this data set support the behavior of the function that we saw in the first data set. However, this function may not be common enough to use as a good metric to measure abnormal behavior on web pages.

Next, we took some different approaches to analyzing the integer functions of this data set. We found that all the dispatch IDs of these functions tended to group up within one of four specific subranges, and each subrange has a particular pattern for the distribution of the dispatch IDs. If a function uses a dispatch ID outside of these ranges, it may be an indication that an object is being used in an abnormal way.

Our analysis of functions with multiple integer arguments did not provide any tangible results. We could not find any significant relationships in the arguments of these functions that could lead to a way of characterizing normality in functions.

Finally, we found one particular function that had a very specific behavior. It always followed a specific object with a particular set of dispatch IDs, and the arguments of the function were always determined by the previous function. This function seems to have a very well defined normal behavior; encountering a call to function that does not match what we saw would be a strong indication of something abnormal happening.

Ideas for future analysis would be to look more into the popular functions of the third result set. These functions should be more common on the internet in general, so they may provide better metrics for a detector for abnormal behavior. Other things to look in to are more patterns in function ordering. By looking at the ordering of one particular function, we found a strong pattern of behavior. Other functions that we have not looked at may also have similar regular patterns of behavior.

Research: Byte String Analysis

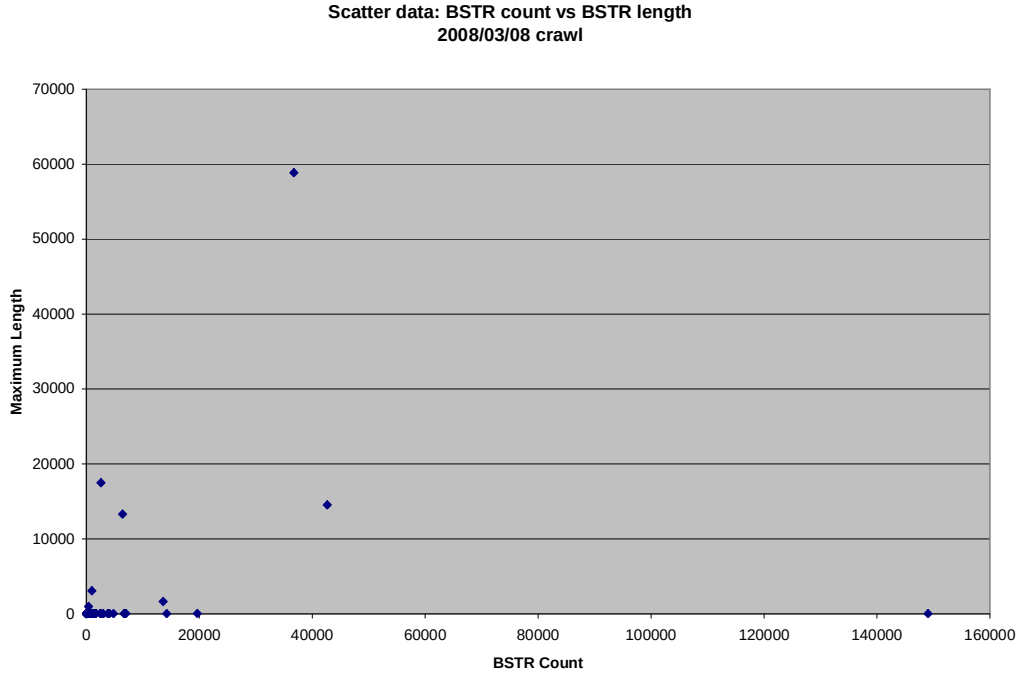
This section addresses the use of byte strings (BSTR) in Javascript code; byte strings are mainly abused for the purpose of performing a buffer-overflow attack to exploit flaws in boundary checking and inject arbitrary shellcode directly into the process. Since a buffer overflow can occur regardless of the content of the string, the length of the string is more interesting, especially since it can be quantified.

We analyze the behavior of Javascript functions that deal with byte strings, and prove that these functions demonstrate general trends in the lengths of strings they are meant to handle. We then perform pruning based on absence of evidence to reveal a much stronger trend for functions that take in byte string lengths. We will also look at data from malicious scripts and see how it compares to the trends we have identified. Finally, we will address how these trends change over time with the state of the Web.

Class-based Trend Analysis

The initial approach to analyzing trends in byte strings derived data from two main crawls. The first crawl, which was performed on February 28, 2008, crawled 5 websites with a very large depth. The second crawl, which was performed on March 8, 2008, crawled 200 websites but only goes through a depth of two links for each website.

For the first analysis, all byte strings were grouped together on a class-by-class basis, where each class is identified by a unique GUID. This gives us a preliminary glimpse of byte string trends but, as we will see later, also has limitations.

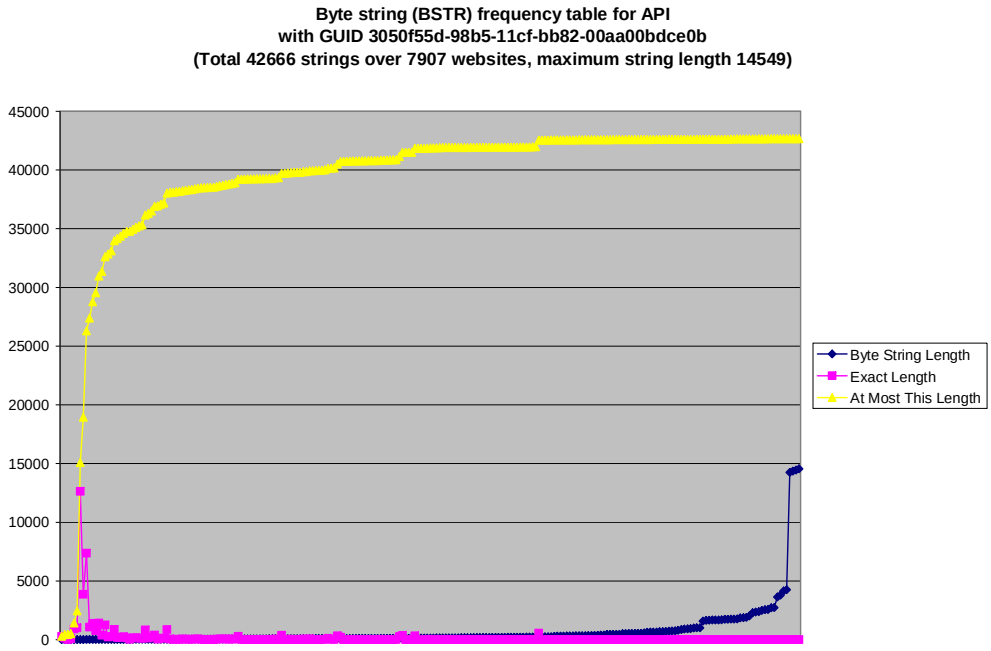


Each point on this scatter plot represents a single GUID seen in the March crawl; the x- and y-axes correspond to how many byte strings that particular GUID saw and the largest BSTR argument that was passed into that function, respectively. More than 70% of the commonly called Javascript classes typically received byte strings of less than

length 20. (39 out of 55 functions from this crawl) Less than 10% of these ever receive a string greater than 5000 characters in length (4 out of 55 functions from this crawl).

The three GUIDs with the most BSTR arguments have varying maximum string lengths; while the most popular GUID never receives strings of length greater than 17, the second most popular GUID receives strings of almost 15,000 characters in length, and the third most popular GUID received strings of almost 60,000 characters in length. All of the other GUIDs tend to fall in the sub-100 range of maximum string lengths, with a handful falling in the 10,000-20,000 range.

Evidence of Javascript classes typically receiving small strings became more apparent when we analyzed all the strings passed into an individual class. As an example, consider the byte strings passed into the class with GUID 3050f55d-98b5-11cf-bb82-00aa00bdce0b:



BSTR length	Exact length	At most this length	BSTR length	Exact length	At most this length
0	287	287	12	1423	30965
1	136	423	13	383	31348
2	20	443	14	1258	32606
3	74	517	15	221	32827
4	907	1424	16	272	33099
5	1014	2438	17	868	33967
6	12638	15076	18	185	34152
7	3865	18941	19	200	34352
8	7362	26303	20	250	34602
9	1079	27382	…		
10	1396	28778	14450	1	42665
11	764	29542	14549	1	42666

The graph plots the length of the byte strings passed into the GUID in increasing order with the frequency of that string and a cumulative total of all strings up to that

length; and the chart below it displays numeric data from a select range of points in the graph. In both of these there is an apparent trend; despite the extremely large size of the largest string passed into the GUID, the typical string length is very small. The largest distribution of strings lied in the 0-20 range, and the very large strings appeared to be abnormalities that did not surface more than a couple of times. This trend repeated itself throughout the most frequently called GUIDs and the GUIDs called with the largest byte strings.

There is an inherent logical flaw, however, in attempting to create a heuristic for normalcy based on classes in that creating a heuristic for a single class imposes that heuristic on all the functions that belong to that class. Since unique functions of the same class may be prepared to accept different byte strings judging all of these functions by the same heuristic could generate very inaccurate results. This analysis was performed during a period where we were still learning the exact role of all the components in the Javascript interpreter, and it became clear that grouping byte strings into sample sets based entirely on the GUID was a flawed approach.

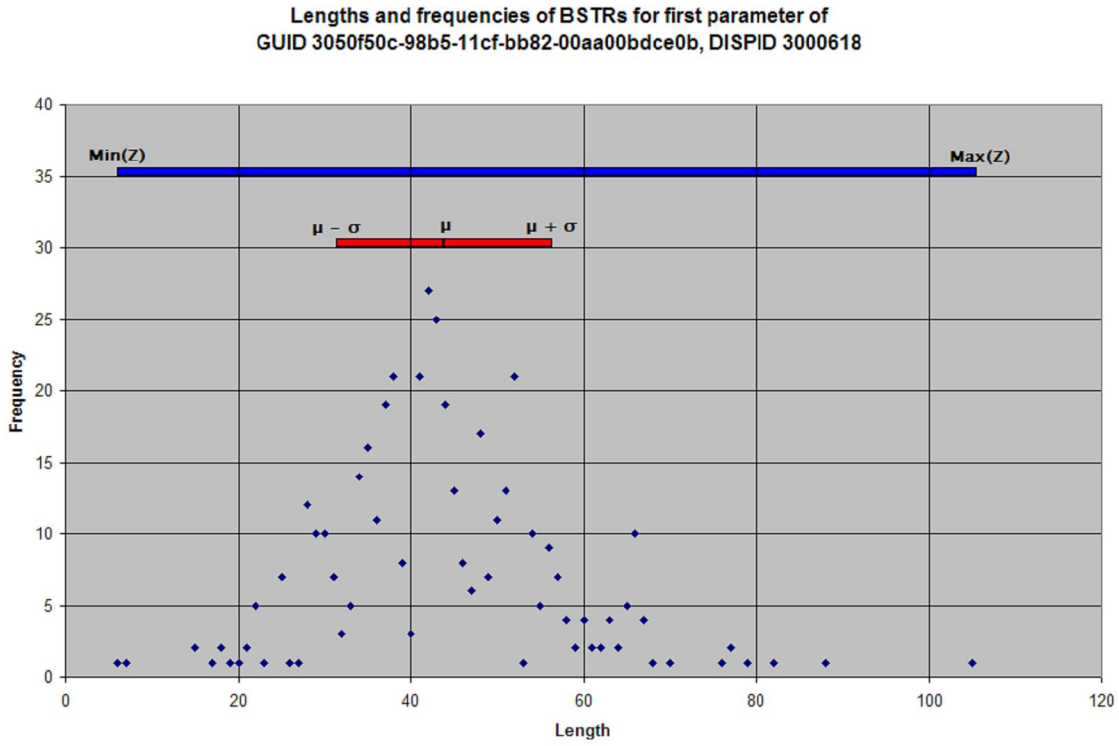
Parameter-based Trend Analysis

After realizing that this methodology was flawed, the research went beyond a class-by-class analysis to an analysis of the byte strings fed into each individual GUID/DISPID pair, which would give us exactly one unique function for every sample set. Furthermore, since each argument in the list may be treated differently, sample sets were further split according to the position of the byte string in the function's parameter list.

Both of these changes greatly increased the size of our sample set, meaning that analysis of individual sample sets was no longer an option; thus, we aimed for a more generalized approach to finding trends in the lengths of byte strings accepted in function parameters. Let us define the following:

- S: The set of all arguments of all unique functions passed into our callback DLL
- Z(X): For some X in S, the data set of all lengths of all byte strings passed into X
- ?(Z): The expected value of Z(X)
- σ(Z): The standard deviation from the expected value of Z(X)

For all X in S, we can calculate the average value and standard deviation of all byte string lengths passed into X. This will give us an interval from (? – σ) to (μ + σ); we expect byte string lengths passed into X to *generally* fall within this interval. However, the meaning of this interval changes depending on the range of values passed into that function. For example, consider the lengths and frequencies of BSTRs passed into the first parameter of the function belonging to the object with GUID 3050f50c-98b5-11cf-bb82-00aa00bdce0b and identified with the DISPID 3000618 from the crawl on May.



This function saw strings from length 6 to length 105 and overall saw as an average string length and standard deviation:

$$\mu = 43.588,\ \sigma = 12.222$$

…meaning we expect values to generally fall into the interval [31.367, 55.811]. The size of this interval, 2σ, is 24.444% the size of the entire interval [6, 105]. Without knowing the entire range into which values fall, there is no way for us to know whether or not byte string lengths typically gravitate towards a specific value in the range of values it receives. Finding the ratio of the expected interval to the entire interval involves the following formula:

$$R = 2\sigma(Z) / (Max(Z) - Min(Z) + 1)$$

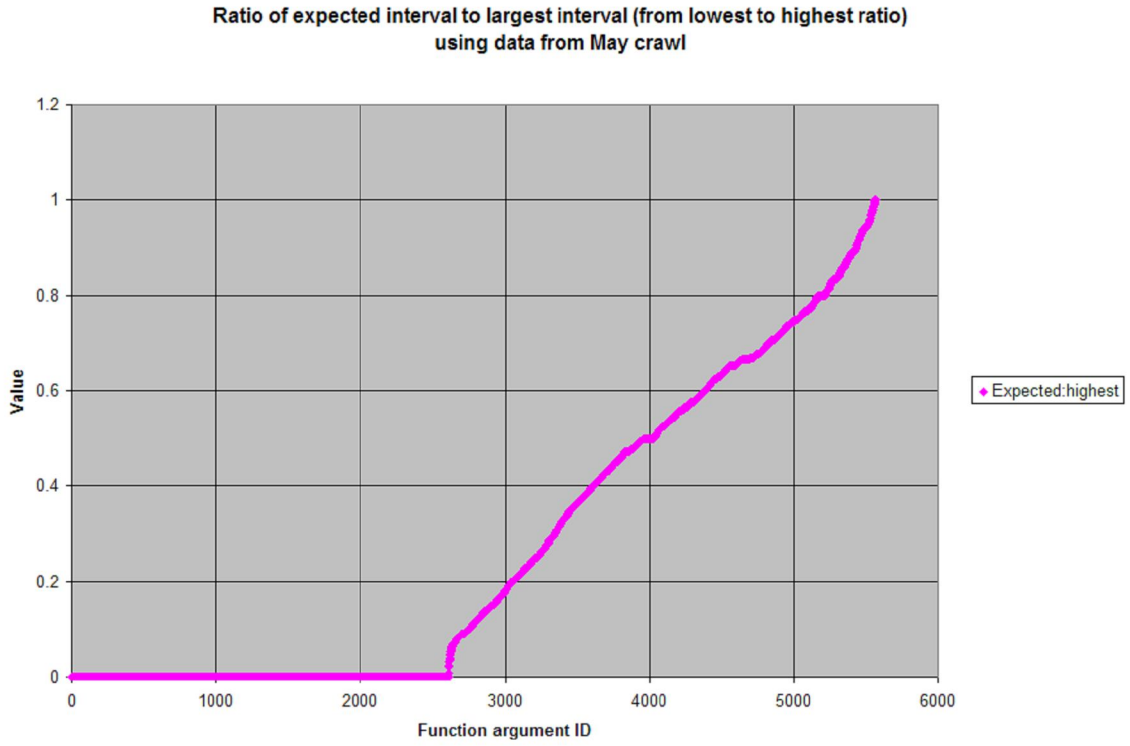
…where $2\sigma(Z)$ is the length of the expected interval $[\mu - \sigma, \mu + \sigma]$; Min(Z) and Max(Z) are, respectively, the lengths of the smallest and largest strings in Z(X); and (Max(Z) – Min(Z) + 1) is the length of the entire interval [Min(Z), Max(Z)] where all the values in the data set lie.

The ratio obtained from this formula will serve as a measure of the definition of a function. For byte strings, we define the definition of a function as the tendency of the function to receive byte strings in a specific range of values. The more defined a function is, the narrower its expected interval becomes in proportion to its entire interval, and the smaller the above ratio becomes. For the above function belonging to the object with GUID 3050f50c-98b5-11cf-bb82-00aa00bdce0b and identified with the DISPID 3000618, then, the ratio is R = 0.24444.

Next, we need to define the criteria for an acceptable ratio, and indicate at what ratio the byte string trend for a specific function is no longer well-defined. We know that the expected interval is always smaller than the actual interval, so $0 \leq R \leq 1$. We also know that when $R = 0.0$, $\sigma(Z) = 0$, meaning that all the byte strings passed into our function were the same length; and we know that when $R = 1.0$, $\sigma(Z) \mathrel{!}= 0$ and the expected interval and the entire interval are equal, though this only happens when the average lies directly between two unique byte string lengths that were passed in at equal frequencies.

As the R value increases from 0 to 1, the standard deviation increases and the sample set becomes less defined. At some point during this transition the sample set starts to resemble a uniform distribution, beyond which it is no longer interesting. See Appendix A for examples of the shape of the sample set as R changes.

From what we observe from the changing curves, a function appears to become well-defined as R goes from 0.5 to 0, so let us define for these observations a well-defined function as a function for which $0 \leq R \leq 0.5$.

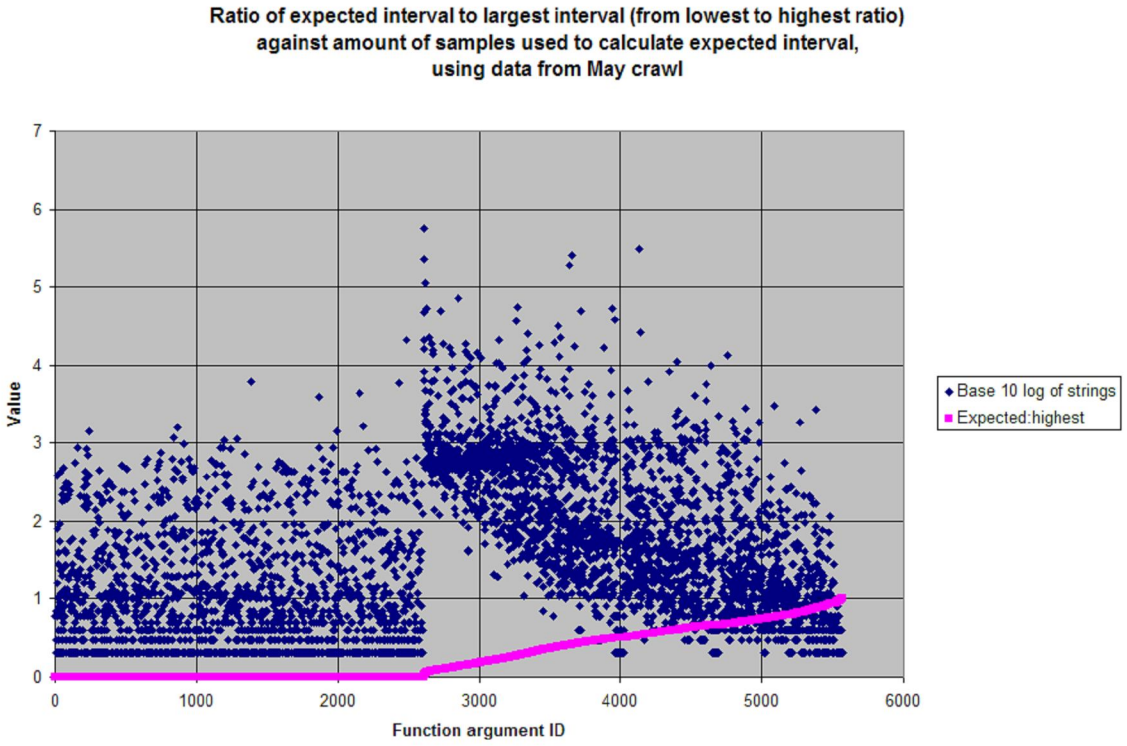


For the May crawl, plotting all the values of R for all function arguments in order from lowest R to highest R appears to yield two distinct curves: (1) a straight horizontal line indicating all the function arguments for which $R = 0.0$; and (2) a curve that is convex for the first half and concave for the second half.

X	Amount of functions for which R ≤ X	Percent of functions for which R ≤ X
0%	2607	0.468127132
10%	2753	0.494343688
20%	3054	0.548392889
30%	3345	0.600646436
40%	3612	0.648590411
50%	**4029**	**0.723469205**
60%	4384	0.78721494
70%	4833	0.867839828
80%	5219	0.937152092
90%	5435	0.975938229
100%	5569	1

The cumulative distribution of the values in this graph shows that 72.3% of the functions in this data set satisfy our criteria for a well-defined function. This number, however, is rather weak given that, of the 72.3% of functions that satisfy our criteria for sufficient definition (4029), 64.7% of those functions had a ratio of 0.0 (2607/4029); and it is even weaker given that most of the functions for which R = 0 are typically seen very infrequently.

This brings us to the next piece of evidence that we will use to measure the byte string definition of a function argument; the relation between the ratio R and the amount of strings passed into the function. For each function argument previously plotted in the graph, we now add the base 10 log of the strings passed into that function argument, so 10 strings would be 1 on the y-axis, 100 strings would be 2, and so on.

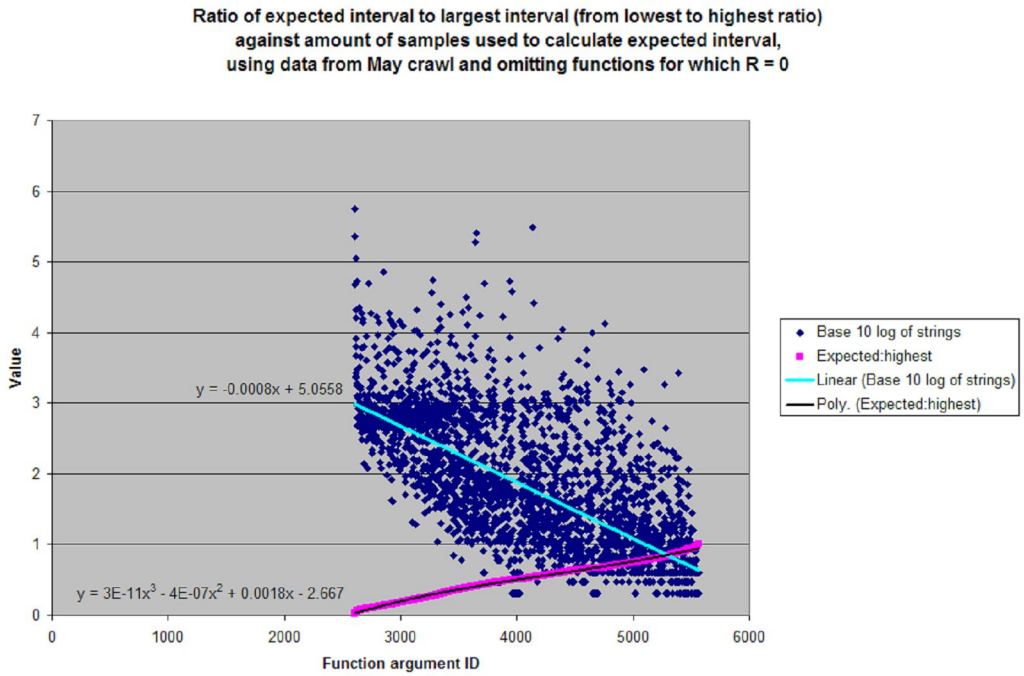


Note that while the byte string frequencies are being plotted on a logarithmic scale, R is still being plotted on the same 0-to-1 scale as before.

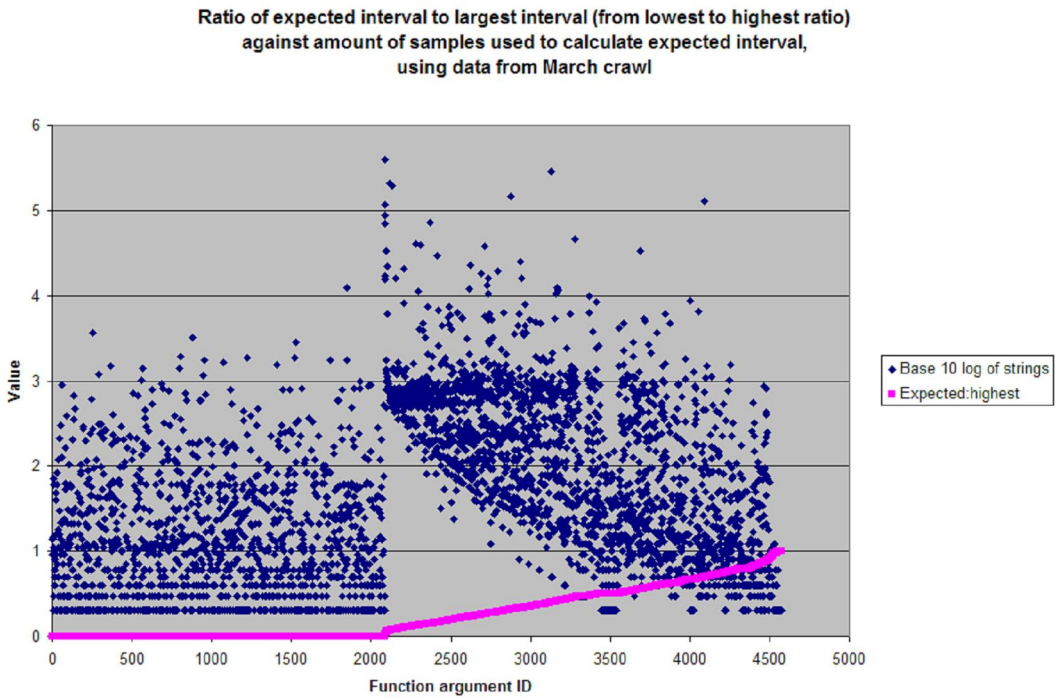
Function arguments where R = 0 typically receive 100 or less strings, with most of them receiving less than 10 strings in total; remember that when the ratio is 0, all the strings passed into the function are the same length. The likelihood of all the strings passed into a function being equal will obviously decrease as the amount of strings passed into the function increases; it follows that string frequency is going to gravitate towards very small values. Large amounts of strings with identical lengths passed into function arguments where R = 0 (a few of these functions receive over 1000 strings of the same length) occur because either (1) the strings expected by that function are usually of a fixed length; or (2) the function is getting called in the same script shared among a large pool of different websites.

The trend in the second half of the curve is much more interesting; as the amount of strings passed into a function argument increases, the ratio tends to decrease! Though Javascript function calls that we typically see very frequently in the web can take in a wide interval of values, most of the values these functions are receiving fall within very narrow intervals.

The next graph attempts to fit the string lengths for functions where R > 0 to a linear curve.



We have pulled a very important observation from this data: *functions with substantial evidence are well-defined in the lengths of byte strings that they expect to receive.* This is an observation that so far has remained consistent over time; if we plot the data from one of our earlier crawls in March in the same manner we observe the same relation between the strings passed into a function argument and the ratio R of that argument:

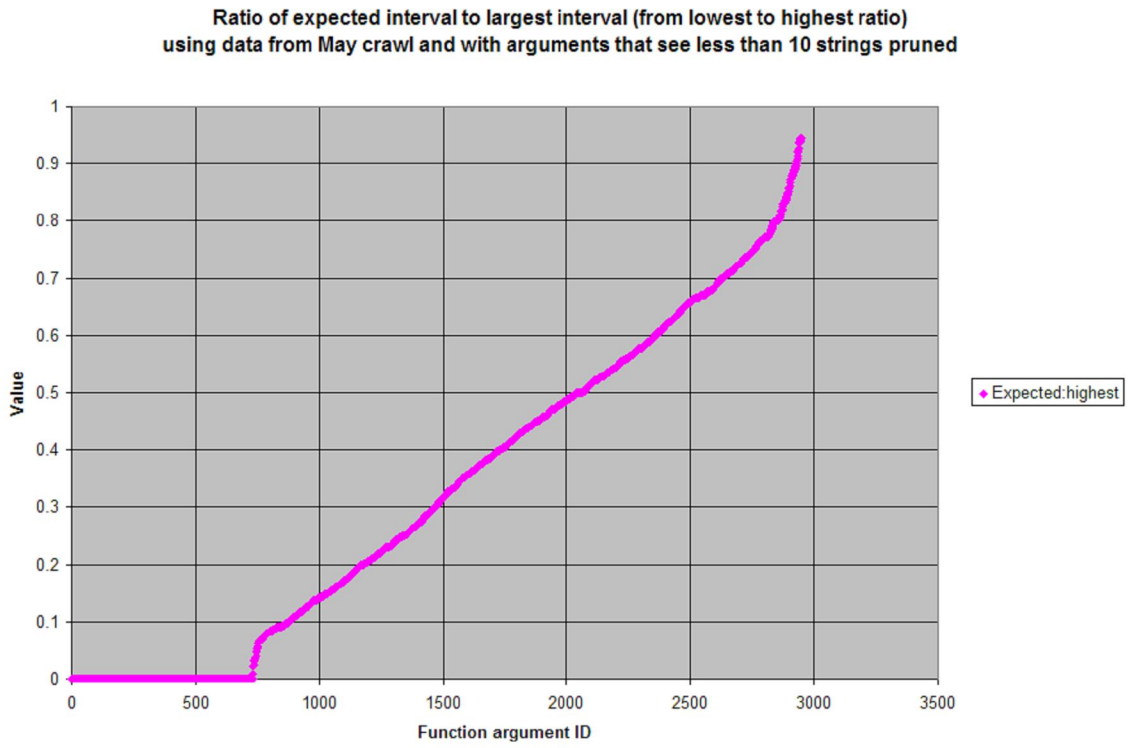


Parameter-based Trend Analysis with Pruning

So far we have identified trends that emerge within function arguments as the amount of evidence – in this case, the total amount of byte strings – becomes increasingly substantial. To strengthen this trend it is necessary to investigate what happens when we begin to remove functions that we identify to have insufficient amounts of evidence.

Suppose, for the sake of argument, that I create a new rule: we cannot say anything about any function argument for which we have not seen at least 10 strings. This means our new plot will ignore any and all function arguments that received at most 9 strings.

The resulting scatter plot and cumulative distributions follow:



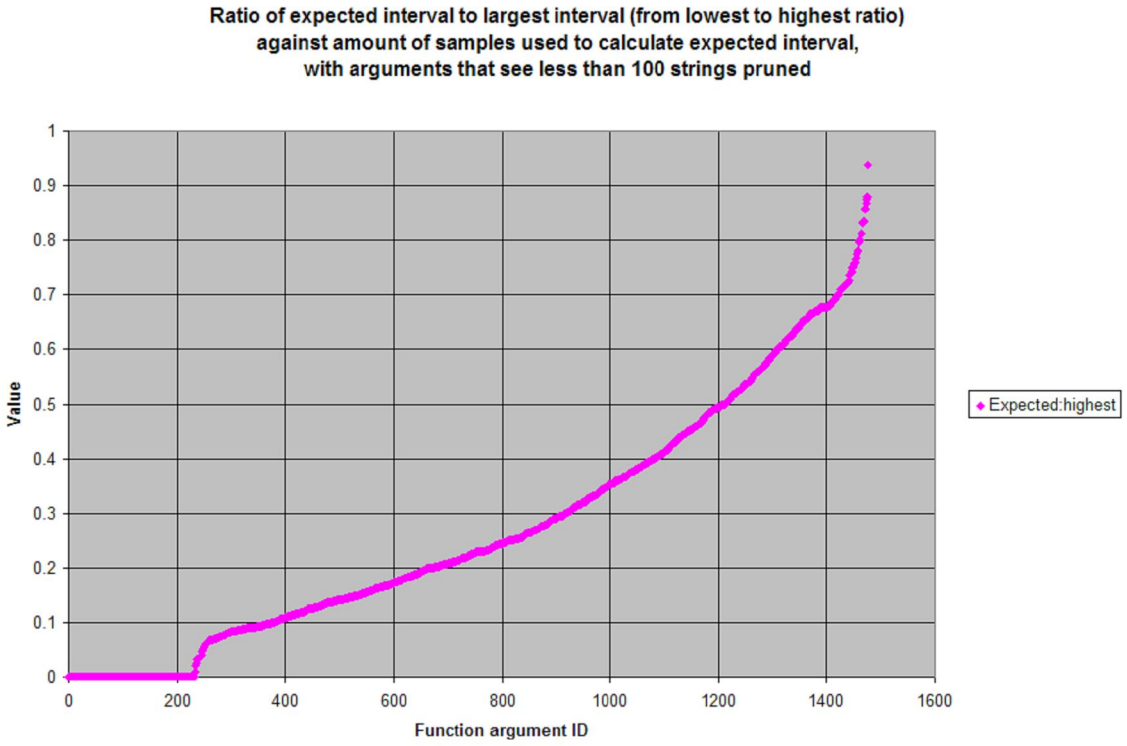
X	Amount of functions for which R ≤ X	Percent of functions for which R ≤ X
0%	731	0.248133062
10%	876	0.297352342
20%	1177	0.399524779
30%	1469	0.498642227
40%	1729	0.586897488
50%	**2063**	**0.700271555**
60%	2355	0.799389002
70%	2630	0.892735913
80%	2855	0.969110659
90%	2929	0.994229464
100%	2946	1

Initially it appears that our trend has weakened, as now only 70% of function arguments fulfill our criteria for a well-defined ratio (as opposed to 72% before the pruning). However, consider the amount of functions for which R = 0. Whereas before the pruning 2607 function arguments existed with a ratio of 0, that number has now been reduced to 731 – 28% of the original amount of zero-ratio functions and 35.4% of all functions with $R \leq 0.5$ (as opposed to the earlier 64.7%).

Also consider the function arguments for which $0.5 < R \leq 1.0$. Prior to performing our pruning, 1540 function arguments had a ratio that fell into this range; after the pruning, 883 functions fell into this range – 57% of the original amount. Then consider that the number of function arguments that fit our criteria for being well-defined decreases from 1422 to 1332 – 93.6% of the original amount. While the quantities of

zero-ratio and poorly-defined function arguments have been severely reduced, the quantity of well-defined function arguments has remained almost unaffected.

This reinforces our earlier observation of function arguments with substantial evidence demonstrating well-defined trends; and this trend becomes even stronger when we increases the threshold of byte strings below which a function argument is pruned from 10 to 100.



After pruning out all function arguments that see less than 100 strings, the amount of zero-ratio arguments decreases from 2607 to 232 (8.9%) and the amount of poorly-defined arguments decreases from 1540 to 266 (17.2%); but the amount of well-defined functions merely decreases from 1422 to 979 (68.8%).

X	Amount of functions for which R ≤ X	Percent of functions for which R ≤ X
0.0	232	0.157075152
0.1	377	0.255247123
0.2	671	0.454299255
0.3	918	0.621530129
0.4	1079	0.730534868
0.5	1211	0.819905213
0.6	1307	0.884901828
0.7	1420	0.96140826
0.8	1463	0.990521327
0.9	1476	0.999322952
1.0	1477	1

Pruning beyond this amount would get rid of most of the functions we ever see, so going beyond 100 strings is fairly meaningless; nonetheless, the results of this function pruning should strengthen our initial observations and serve as substantial proof for the existence of trends and tendencies in the lengths of byte strings received by individual function arguments.

Prune if total strings is less than	1	10	100
R = 0.0	2607	731	232
0.0 < R ≤ 0.5	1422	1332	979
0.5 < R ≤ 1.0	1540	883	266

Trend Analysis with Malicious Data

The focus of our research has entailed the investigation of a heuristic for normalcy of Javascript behavior to later use as a judge of whether or not a function is behaving abnormally; it follows that any trend we find for distinguishing normal data from abnormal data should be capable of distinguishing non-malicious data from malicious data in the same manner.

For the purpose of comparing the behavior of malicious scripts with the behavior of benevolent scripts, Symantec provided us with a set of test scripts used to check the signatures of their Browser Defense technology (also referred to as the Canary engine). These are non-malicious scripts that can be used to test whether or not a security system defends against a browser exploit; though they perform the attack necessary to compromise the browser they do not inject any actual shellcode into the process space, so the worst thing these test samples can do is cause the browser to crash.

Our main testing environment was Windows XP in Internet Explorer 7, so we investigated a small set representative of the types of vulnerabilities in XP IE7 that were open to a hostile attack. Exploit behavior would fall into one of three categories:

- Passing in an unexpected byte string (BSTR) to a function
- Passing in an unexpected integer (I1, I2, I4, UI1, UI2, UI4) to a function
- Passing in an unexpected object (DISPATCH) to a function

Though many of these attack samples exploited GUIDs that we had yet to see in the wild, implying that these functions should rarely be used, there were a small set of samples for which the GUID/DISPID call that was responsible for the attack was to a function that we saw in the wild very frequently. We will look at data from one of the functions that user byte strings to trigger an exploit in a function we see very frequently.

The sample in question, when run with Canary enabled, triggers a signature called "MSIE Popup Window Address Bar Spoofing Weakness". According to the Antivirus Center at Symantec's official website:

> "This issue occurs because it is possible to display a popup window with only a portion of the address bar initially displayed to the user. By using a combination

of special characters in a URI that launches a popup window, an attacker can cause the popup to appear to derive from a trusted site by directing primary focus to only a specific portion of the originating URI. This will cause the address bar to initially display the URI of a trusted site, which the content may display attacker-controlled data." [7]

We ran the test script through our callback DLL and found that the following function call triggered the exploit:

DISPID	GUID	Params	Type 1	Value 1
1026	3050f55f-98b5-11cf-bb82-00aa00bdce0b	1	BSTR	150

According to our data from the crawl in May, the function belonging to GUID {3050f55f-98b5-11cf-bb82-00aa00bdce0b} with DISPID 1026 was fed a byte string argument 491 times over the 20,416 websites visited during that crawl. Of those 491 strings, the smallest string was length 70, the largest string was length 80, and the average string length was 76.32 with a standard deviation of 2.33. Not only does the length of this exploit byte string lie nowhere in the expected interval [73.99, 78.65], but it also lies nowhere in the entire interval [70, 80], and in fact its length of 150 is 31.6 standard deviations away from the average length!

For this particular function call, then, it appears that our gathered data can successfully be used to distinguish normal scripts from abnormal, malicious scripts. The rest of the samples that did use BSTRs to trigger a buffer overflow exploited functions in GUIDs that we had never seen in our entire crawl, which can mean one of two things: either that function is *always* abnormal when called with byte strings, or the calls to the function are so rare that we would need to perform a very broad crawl to find one. From this observation, nonetheless, it stands that we can define trends for byte string behavior and we can distinguish normal and abnormal scripts using this trend.

Trend Volatility

Live websites, often being operated manually by one or more webmasters, can constantly change which scripts they use based on what script-based functionality the webmaster chooses to add or remove. Do these changes ultimately affect our gathered data, and if so to a controllable extent?

To answer this we performed 28 crawls over the same 1000 websites from May 7 to May 13 and compared the callback data gathered from each crawl. The chart below gives a rough overview of the data collected from each of the 28 runs.

Run	Size (KB)	Size (MB)	DLL calls	URLs w/scripts
1	151356	147.81	2071719	469
2	157178	153.49	2155154	461
3	153634	150.03	2091515	466
4	162288	158.48	2236706	465
5	153640	150.04	2091038	465
6	157247	153.56	2148058	463
7	152411	148.84	2074383	463

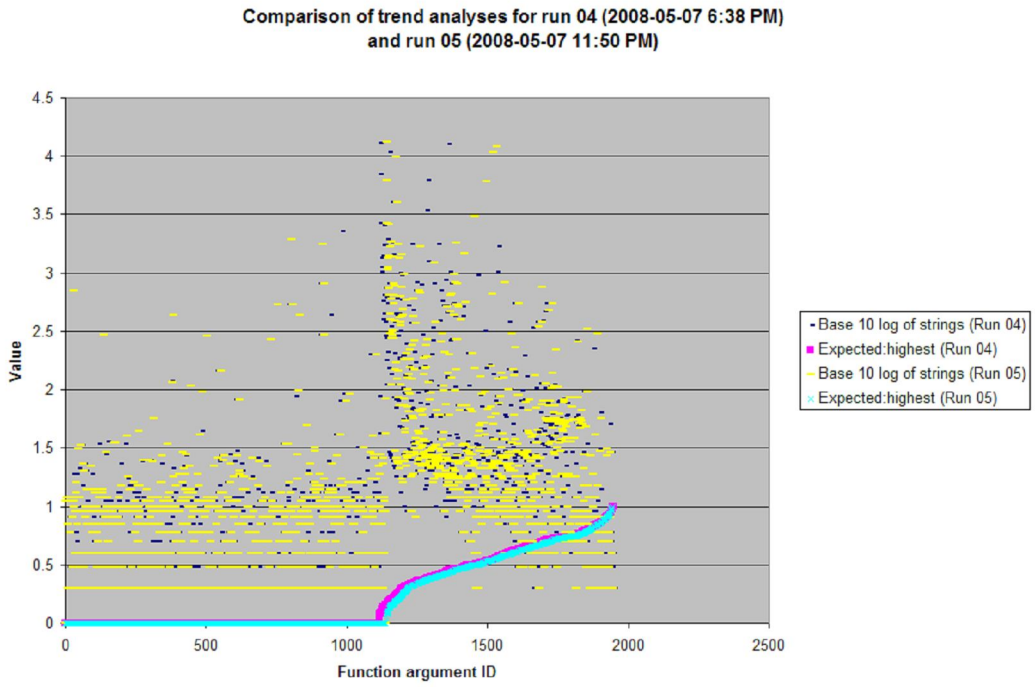
8	152776	149.20	2082031	463
9	143086	139.73	2041672	465
10	158285	154.58	2172353	466
11	160603	156.84	2213557	465
12	153172	149.58	2096330	463
13	165713	161.83	2304933	465
14	153186	149.60	2164695	461
15	157522	153.83	2166498	464
16	163440	159.61	2263828	466
17	139088	135.83	2015998	462
18	159899	156.15	2219125	463
19	161024	157.25	2220763	465
20	160108	156.36	2211366	464
21	149148	145.65	2090184	461
22	161600	157.81	2232686	466
23	162083	158.28	2236013	465
24	155612	151.96	2189636	465
25	154043	150.43	2176079	463
26	160625	156.86	2221586	463
27	157313	153.63	2173366	464
28	158556	154.84	2199971	461

The first column indicates the run that was analyzed, and the next three columns indicate the size of the data collected from the run, in terms of kilobytes, megabytes, and the raw amount of function calls intercepted by our callback DLL. What is most noticeable is that the size of data collected from the run varies wildly, with many data sets differing in size by at least several megabytes and several hundred thousand DLL calls. There does not appear to be a consistent pattern, either; the size of the data set doesn't appear to have any relation to the time of day at which the crawl took place, and in fact the changes in size are somewhat random.

On one hand, a difference of several hundred thousand DLL calls is not a change of tremendous magnitude when you consider that each crawl called the DLL at least two million times; however, it is a significant change that needs to be taken into account.

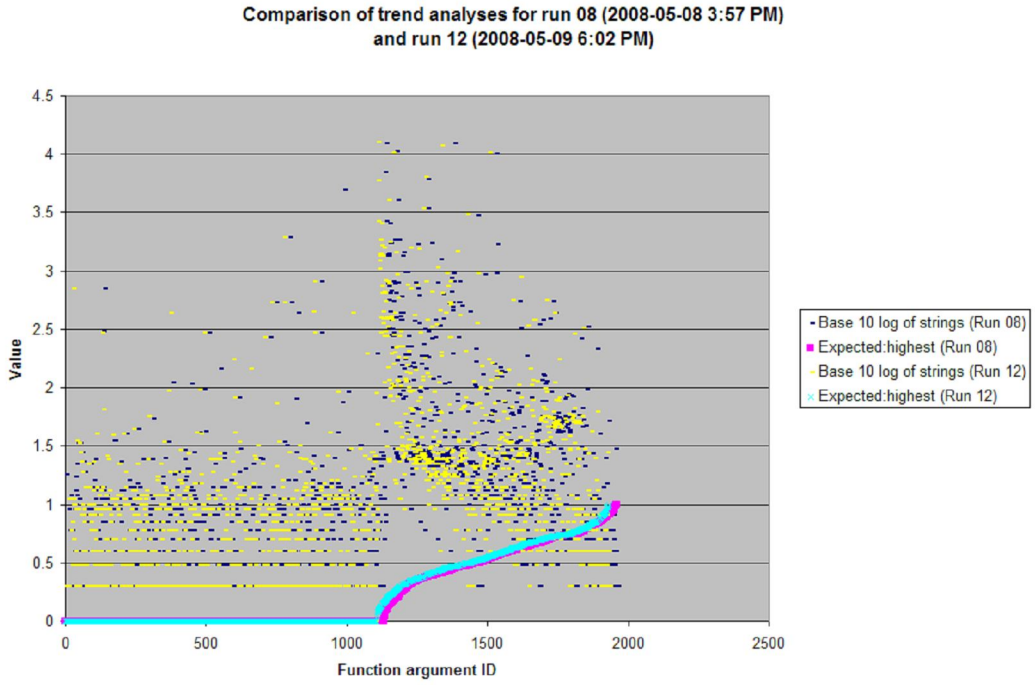
The last column, which lists the amount of sites out of the 1000 sites crawled that actually contained scripts for our callback DLL to intercept, is also interesting as even this value does not remain consistent. For that matter, it doesn't even consistently decrease; from runs 9 to 11 we go from 465 sites up to 466 and then back to 465 again, and from runs 13 through 15 we go from 465 sites down to 461 and then up to 464. Again, this is not a tremendous change, but it's still enough of a change that it could affect the trends we see in Javascript function calls.

This overview provides a very basic picture of the changes in Javascript site activity over time; however, it is also important to analyze the changes in trends seen from each day. The initial trend analysis involved plotting the ratio R of the size of the expected interval to the size of the entire interval for our May crawl across 20416 websites; this method of analysis will now be repeated with selected runs from the set of 28 crawls to analyze how the gathered data changes.

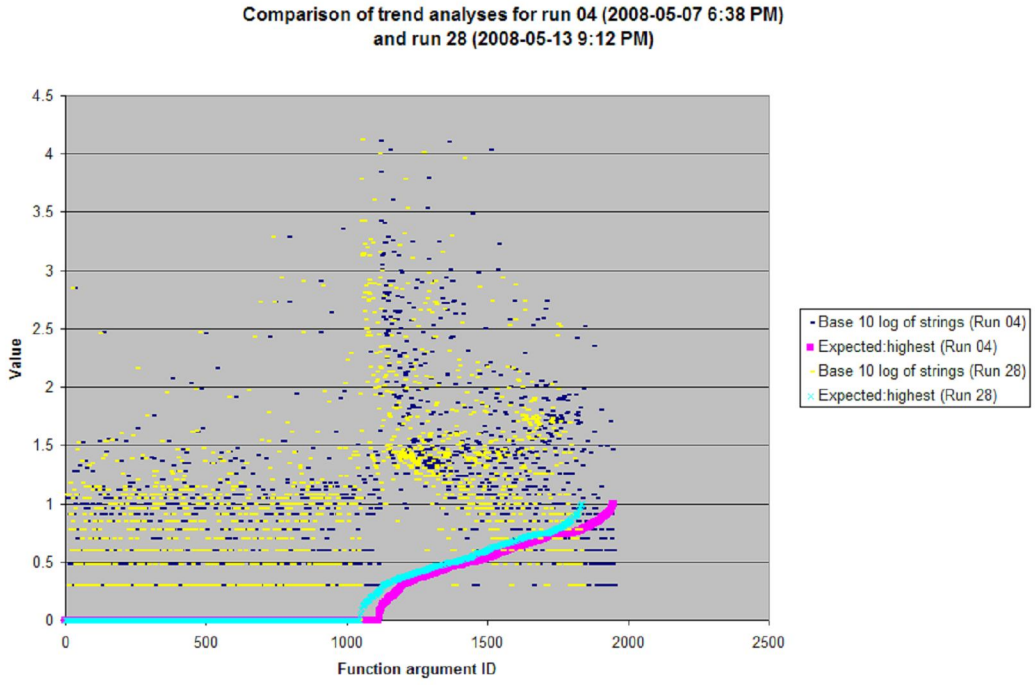


First, we compare the ratios from runs 4 and 5, which occurred approximately five-and-a-half hours apart. Both crawls saw 465 websites with scripts, but the data gathered from run 4 is 158.48 MB whereas the data gathered from run 5 is 150.04 MB. The ratio plot overall changes very slightly; the major change appears to be in the increasing amount of zero-ratio argument from run 4 to run 5 and the decreasing amount of arguments where $R > 0.5$.

We then compare the ratios from runs 8 and 12, which occurred approximately a full day apart. Both crawls saw 463 websites with scripts, and run 12 only saw a marginally higher amount of data. The ratio plot also changes slightly, but this time the change is different; the zero-ratio arguments decrease, and the remaining arguments have a higher tendency of yielding a ratio close to $R = 0.5$.



Finally, we compare the ratio plots of runs 4 and 28, which occurred roughly 6 days apart. Run 4 saw more sites with scripts and a larger data set than run 28. Again, the zero-ratio arguments decrease, and the second half of the curve gravitates towards a ratio of R = 0.5.



These analyses confirm that the state of Javascripts on the Internet does fluctuate very frequently. Though these changes are small enough in magnitude that any heuristic developed at one point in time could with a large enough sample set remain stable for a period of time, these changes do exist and need to be recognized as they will affect the accuracy of the heuristic in question.

Conclusion

There are several trends we can identify for function arguments that accept byte strings. Function arguments do tend to receive byte strings with lengths in a specific interval, and this interval becomes increasingly defined as we see more instances of this function used in the Web. From what we have seen from exploits against Internet Explorer 7, defining these intervals of normalcy is successful in distinguishing between normal data and malicious, abnormal data; and even exploits that use byte strings to trigger unseen GUIDs have suspiciously large byte string values, indicating that with a deep enough crawl we can obtain substantial evidence on nearly any Javascript function call. This approach is not limited to byte strings, either, but can be applied to any argument type that can be quantified as a number.

A heuristic for normalcy of byte strings almost certainly exists on an argument-by-argument basis; however, since Javascript activity over the Web is subject to change over time, any trends we identify will always have an element of volatility. Any heuristic for normalcy that is to be robust will therefore need some learning capability to be able to adapt to situations where a normal string is accidentally flagged as abnormal because the heuristic for normalcy is out of date.

The next step beyond this is to design an exact, robust heuristic for normalcy and at the same time develop a learning algorithm to adapt this heuristic to the ever-changing state of the Web; and since, even after pruning functions with insufficient evidence a few outlier arguments exist with poor ratios of expected to entire intervals it may be necessary to develop a means of deciding when this heuristic for normalcy can actually be determined.

Research: Activity Analysis

To properly analyze a function's behavior, it is important to note how that function is used by different sites. While it is obvious that each function has its own defined characteristics, why that specific function was called is somewhat of a mystery. This section attempts to analyze that caller-callee relationship between a site and the script functions it calls.

There are multiple ways to differentiate sites' behavior from one another, but not many are quantifiable. One could attempt to find a pattern in the source code of each site, but much of that is not available to the client. Another path could have been to analyze each function per-domain, stipulating that the same set of developers worked on those sites so they should display a function usage pattern. The issue is that those two, as well as many other methods, are not easily quantifiable by number. To be able to analyze sites, the defining characteristic chosen was a site's script activity. Each function in this section is measured against the number of function calls the site calling it made.

Activity Categories

A script was written that broke down the number of function calls per site. Using this script, it was determined that the function call data had some interesting characteristics.

Average	Std Dev	Median	1st Quartile	3rd Quartile	Interquartile Range
5777	14181	1456	438	4029	3591

Also, there were many outliers in the data (roughly 14%), with some sites having more than 400,000 function calls. Examining this data, it becomes clear that it is very "top-heavy" and that there are sites that are created quite differently from the norm.

To establish a good balance between the number of sites in each category and the total function calls in each category, the median/quartile observations (i.e. a box and whisker plot) were used. This guaranteed a certain number of sites in each category, as the median establishes the halfway point in the data. It was decided that there would be three categories, so that relationships could be more easily established. The higher limit for the lowest category was established at 1500, so that slightly more than 50% of the sites would make up the category. The higher limit for the middle category was established at 5000, so that around 28% of the sites would wind up in the category. The last category would then also include around 20% of the data.

It is a topic of further to research to examine how the decision of how to place the dividers, as well as the number of categories, affects the distribution amongst the functions.

Category Heuristic

A heuristic was designed to characterize what category each function was more closely associated with

A script was written that broke down how many times each function was called in each category, and using this data another script was written that would find the heuristic value for each function. The exact function used is:

```
F =((avgl - avgsite)*(L - avgfunc)+(avgm - avgsite)*
(M - avgfunc)+(avgh - avgsite)*(H - avgfunc)) / 3
```

Where avgl, avgm, and avgh are the average number of function calls per category (542, 2882, and 22745 respectively), avgsite is the overall average number of function calls per site (5777), and avgfunc is the avg number of function calls per function (1984). L, M, and H are the specific number of times the function was called in the low, medium, and high category.

Using this metric, each function had a different value that would inform whether it fit more in the high, medium, or low category.

Statistical Variation

The heuristic script generated a value for each function and sorted them by this value. The corresponding most positive and negative functions were chosen to analyze the differences in argument type frequencies and argument value distributions between the categories. The functions that had the most negative value were those that appeared most in the lower-valued categories and not the higher-valued categories, while the ones that had the most positive value were those that appeared most in the higher-valued categories and not the lower-valued categories.

The functions analyzed were the following:

GUID:DISPID (i.e. which specific fn)	Heuristic value
3050f502-98b5-11cf-bb82-00aa00bdce0b:-2147417111 (first)	8800912345
3050f50c-98b5-11cf-bb82-00aa00bdce0b:-2147417111 (second)	6709843885
3050f523-98b5-11cf-bb82-00aa00bdce0b:-2147417111 (sixth)	4638170246
3050f548-98b5-11cf-bb82-00aa00bdce0b:-2147417111 (eigth)	2609868001
3050f55d-98b5-11cf-bb82-00aa00bdce0b:3000099 (tenth)	2379275418
3050f502-98b5-11cf-bb82-00aa00bdce0b:-2147412104 (twelfth)	2264036428
3050f502-98b5-11cf-bb82-00aa00bdce0b:3000029 (sixteenth)	1903251257
3050f38e-98b5-11cf-bb82-00aa00bdce0b:0 (bottom fourth)	-33721359
3050f55d-98b5-11cf-bb82-00aa00bdce0b:10006 (bottom third)	-34333777
3050f51c-98b5-11cf-bb82-00aa00bdce0b:1002 (bottom second)	-71323345
3050f51c-98b5-11cf-bb82-00aa00bdce0b:-2147418043 (bottom first)	-78751033

There were actually functions that sat between some of these, but they did not have any arguments associated with them, so they were not analyzed. The functions will be referred to as (in order, going down the table): first, second, sixth, eighth, tenth, twelfth, sixteenth, bottom fourth, bottom third, bottom second, bottom first. These names were chosen because they are each function's location in the heuristic data.

A script was written that grabbed the function calls that involved only those functions listed above. This allowed for quicker access time.

The first observation made is that, at the higher heuristic value, functions are passed fewer arguments than those functions whose heuristic values were more negative. As mentioned before, several functions were not considered for analysis because they were never, in the data set, passed any arguments. In addition, the frequency of 0-

argument vs. 1+ arguments for the functions that had the highest heuristic value was strikingly different. For example, for the first function:

LOW:	MID:	HIGH:
0 arguments: 12790	0 arguments: 138095	0 arguments: 1581416
I4 arguments: 0	I4 arguments: 0	I4 arguments: 0
BSTR arguments: 736	BSTR arguments: 1394	BSTR arguments: 3708
DISPATCH arguments: 0	DISPATCH arguments: 0	DISPATCH arguments: 7
NULL arguments: 0	NULL arguments: 0	NULL arguments: 0
BOOL arguments: 0	BOOL arguments: 0	BOOL arguments: 0

While the bottom first function had the following distribution:

LOW:	MID:	HIGH:
0 arguments: 1850	0 arguments: 44549	0 arguments: 1367
I4 arguments: 0	I4 arguments: 0	I4 arguments: 0
BSTR arguments: 2921	BSTR arguments: 32945	BSTR arguments: 441
DISPATCH arguments: 0	DISPATCH arguments: 0	DISPATCH arguments: 0
NULL arguments: 0	NULL arguments: 0	NULL arguments: 0
BOOL arguments: 0	BOOL arguments: 0	BOOL arguments: 0

The first thing that jumps out is that the bottom first function is called more in the lower categories than the higher categories, while the first function is called more in the higher categories, which is to be expected. The other thing to notice is the ratio of times the function is called with arguments as opposed to without. First is called significantly more times without arguments, especially in the high category, while bottom first is called with BSTR arguments a lot more often.

Another script was written to grab the distribution of each argument type for the functions. The script was given as an input which function (first, second, etc.) and what argument type, and then it analyzed the distribution of argument values for the given type. This script currently only analyzes the arguments individually, it can be expanded to look at a tuple of arguments, or look at sequence of function calls, but that is a topic for further research.

The function that had the biggest sample size for analyzing argument distributions in the negative heuristic value range was bottom fourth. The argument type frequency looks as follows:

LOW:	MID:	HIGH:
0 arguments: 29636	0 arguments: 15401	0 arguments: 5679
I4 arguments: 9882	I4 arguments: 6766	I4 arguments: 6532
BSTR arguments: 0	BSTR arguments: 0	BSTR arguments: 0
DISPATCH arguments: 0	DISPATCH arguments: 0	DISPATCH arguments: 0
NULL arguments: 0	NULL arguments: 0	NULL arguments: 0
BOOL arguments: 0	BOOL arguments: 0	BOOL arguments: 0

Several similarities can be seen between the categories in this function. The first striking similarity is that, out of the more than 20,000 times it is called, there are not very many differing arguments that are passed to it. No more than 50 different integers are passed into the argument, and that holds across each category. The other similarity is that most of the time, the function is passed lower numbers.

Value	Low	Mid	High
1	5998	2650	2846
2	1750	1831	1811
3	1750	1831	1811

This constitutes a majority of all of the arguments passed into the function in each category. The percentages are a little different in each category, but the general trend holds true.

There are, however, differences. Each category has its own set of spikes. A spike will be defined here quite loosely, as just an abnormal frequency in the data for one category that does not exist in others.

```
Spikes in low:
11,40
20,45
42,40
64,55
150,51
```

```
Spikes in mid:
20,74
64,174
131,54
264,96
```

```
Spikes in high:
None
```

These spikes look more like anomalies in the data than actual trends. They could result from a single site using this specific function in a certain way, for example. They also constitute less than 5% of the data in each category, so no real conclusion can be drawn from them.

The function that had the biggest sample size for analyzing argument distributions for highest positive heuristic values was second. The argument type frequency looks as follows:

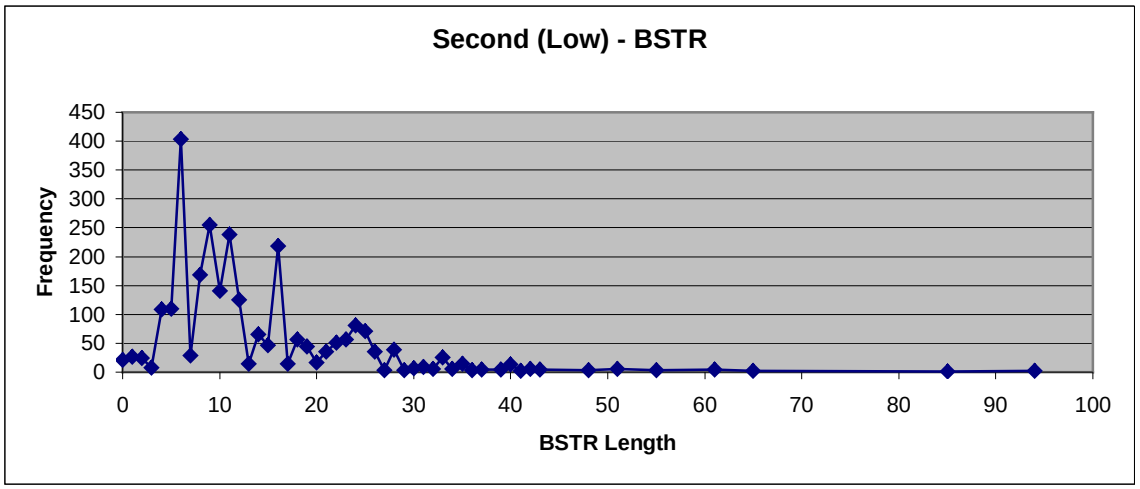
LOW:
0 arguments: 20713
I4 arguments: 0
BSTR arguments: 2634
DISPATCH arguments: 14
NULL arguments: 0
BOOL arguments: 0

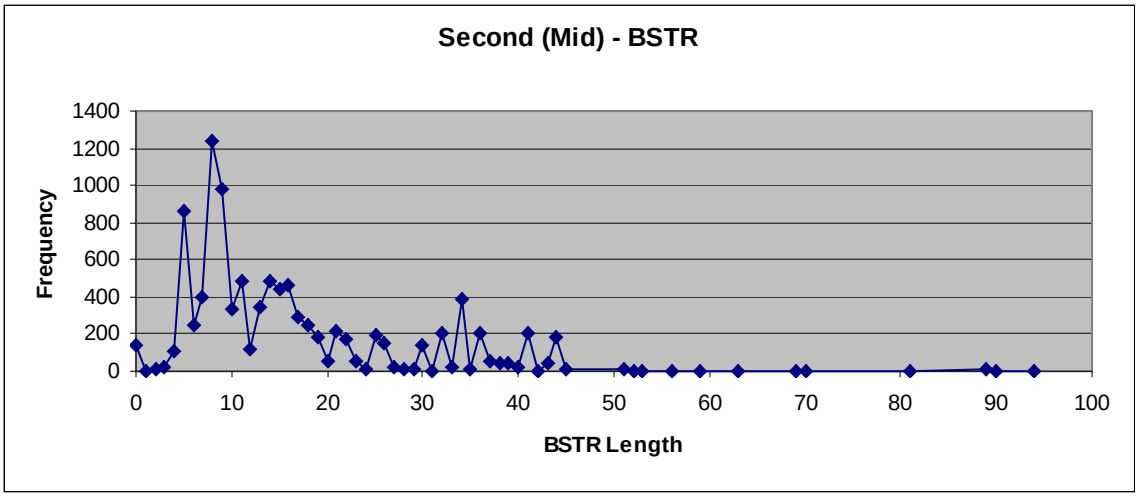
MID:
0 arguments: 170861
I4 arguments: 0
BSTR arguments: 9888
DISPATCH arguments: 1
NULL arguments: 0
BOOL arguments: 0

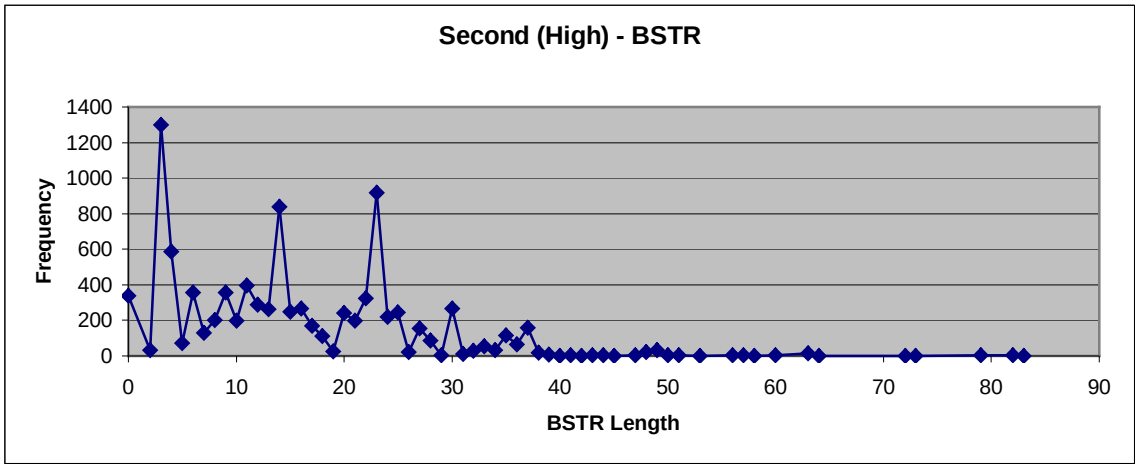
HIGH:
0 arguments: 1215964
I4 arguments: 0
BSTR arguments: 9447
DISPATCH arguments: 19
NULL arguments: 0
BOOL arguments: 0

This function also demonstrates the previously mentioned curiosity of functions that have high positive heuristic values, in that the number of times it is not passed an argument is far greater than the number of times it is passed an argument.

Looking at the charts for the distribution of BSTR length in the second function, there are both similarities and differences. The two main similarities are that much of the data is between the 0-20 range, and there are very few BSTRs encountered of length > 50.







Besides these two similarities, there are some interesting differences. The highest point for the high graph occurs with BSTRs of length 3. This value barely appears in the

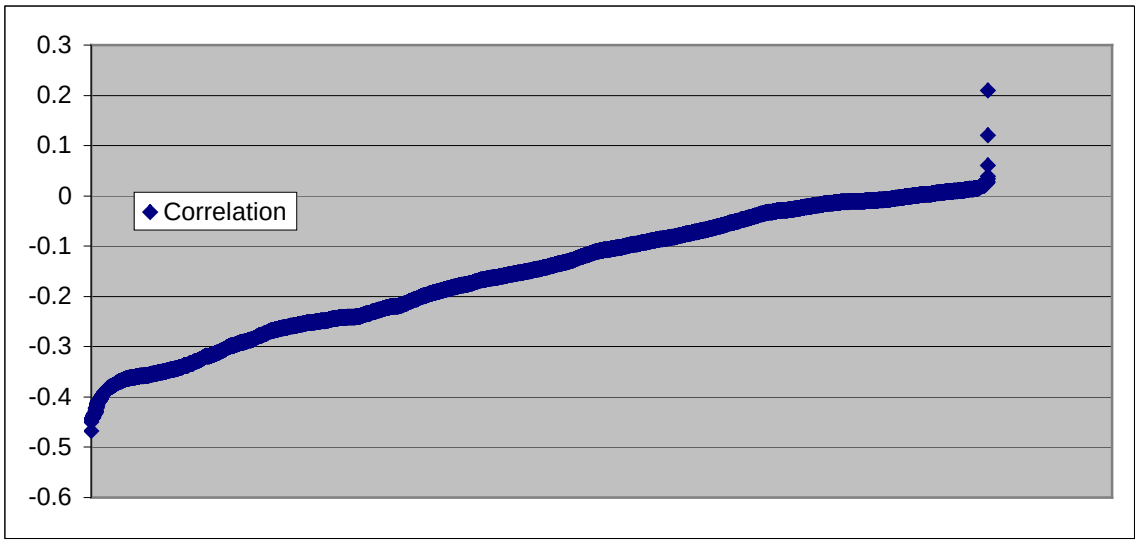
other two categories, showing up 8 times in low and 25 times in the middle categories. The other major difference is that in the low category, this function barely has BSTRs of lengths 30-40, while they appear very common in the middle and high categories. There is also a large spike in the high category at value 23, and this spike does not exist in the other categories (this BSTR length appears 57 times in low, 54 times in middle, and 916 times in high).

The conclusion that can be drawn here is that, for this specific function, BSTR lengths of 0-15 are common in all three categories, but those of length 15-40 are significantly more common in the middle and high categories.

Site Activity and Function Correlation

To be able to calculate the correlation between a site's script activity and the functions it calls, the data had to be separated into each individual function and how active the site calling it was. This produced a set of tuples for each function of the form (number of times called at a particular site, total number of function calls that site made). In the correlation analysis, functions were only examined if they had more than 100 sites calling them, so that there would be enough data to draw a good conclusion.

The correlation values ranged from as low as -.47 to as high as .21.

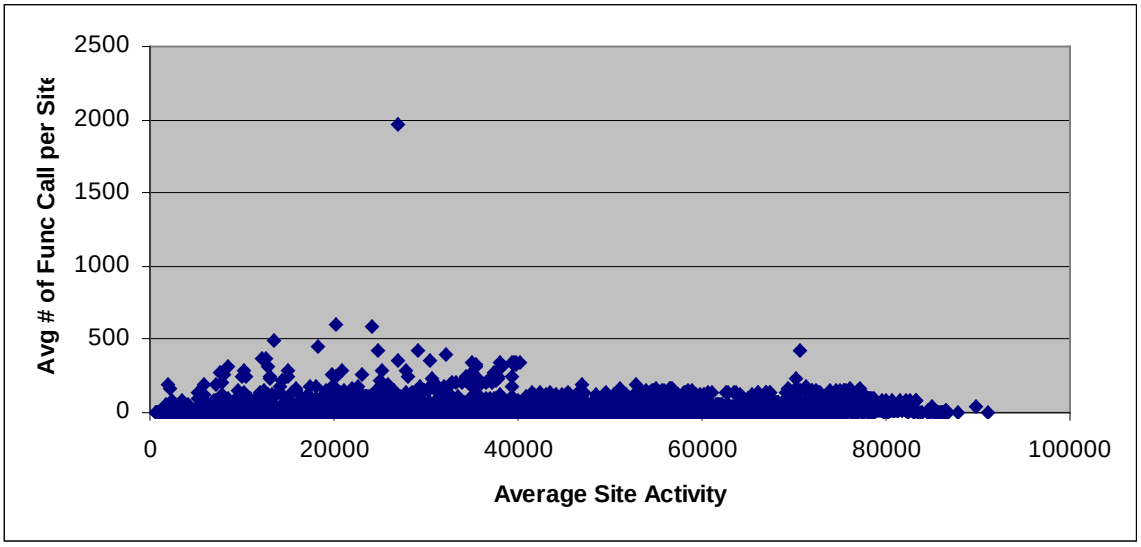


Most functions had a negative correlation, meaning that the more function calls the site made, the fewer times that specific function was called. To further examine this trend, the average number of times a function was called per site was calculated, and measured against the average number of function calls made at sites calling it.

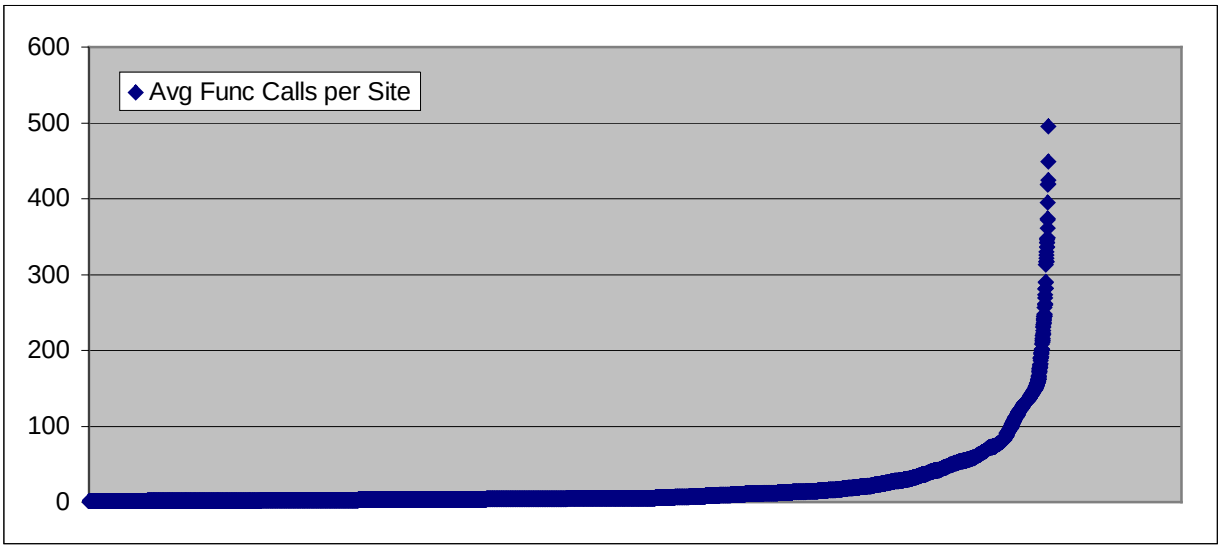
Function Calling Trends

The higher average function call per site, surprisingly, does not climb as sites increase their script activity. This indicates that very active sites are calling a variety of

functions, rather than calling a select few many times.

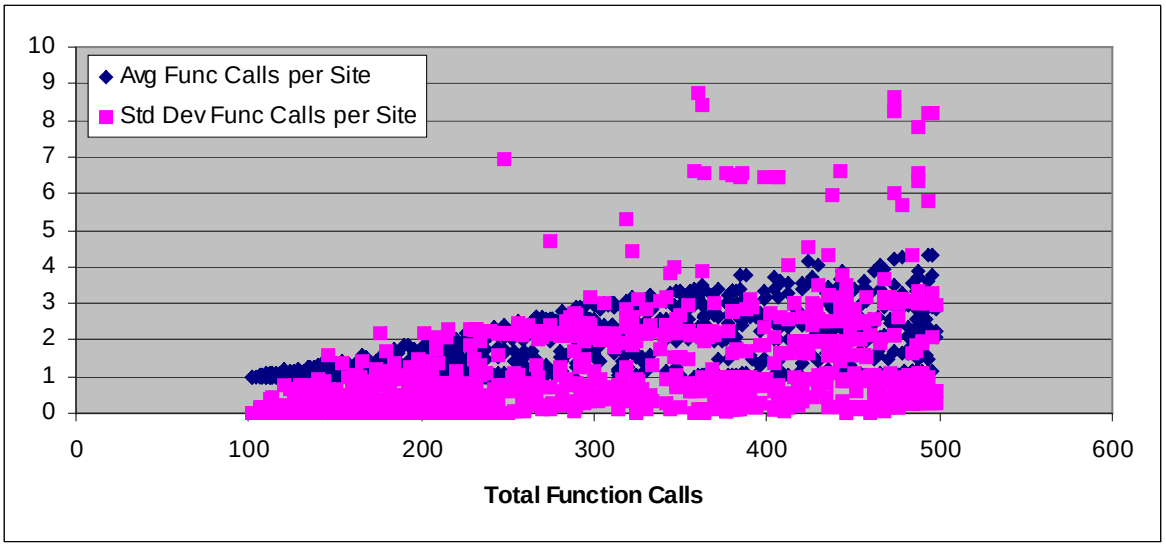


This seems to imply that, in a normal situation, a function is not called more than a couple times at a site. Looking at the data, 54.8% percent of functions are called, on average, less than 5 times per site.

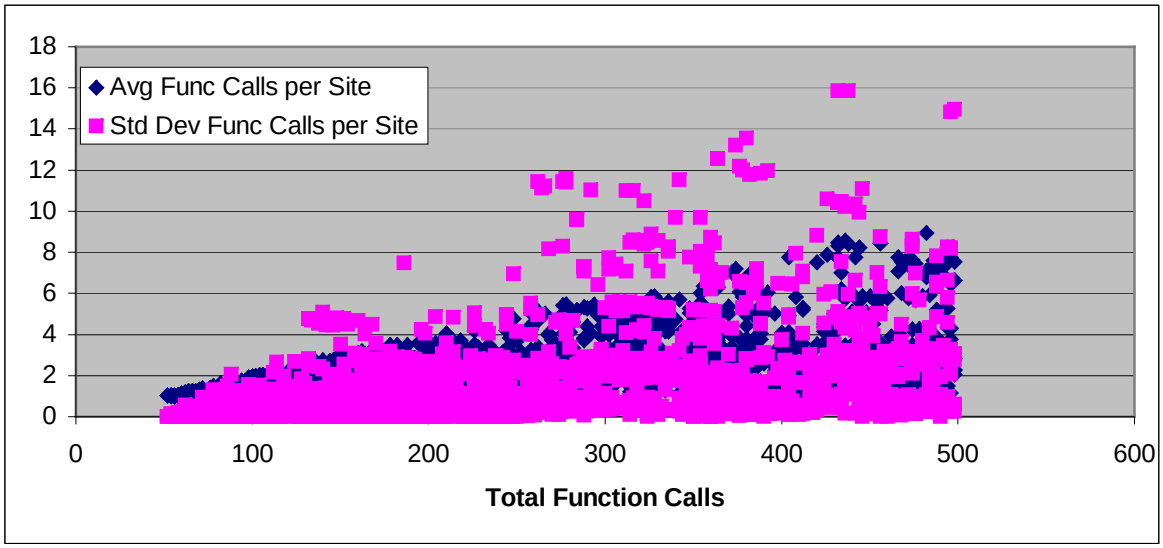


This yields a telling conclusion. The function calls being examined have certain characteristics that can be understood, but what would be the case if a malicious script attacks a weakness in a function that has not been well analyzed or even has never appeared in the data before? The trends seen seem to imply that there are certain elements that should be true regarding unknown functions, especially that this new function will not be called many times. As the total number of function calls increase, the average function calls per site increase, which can be seen in the next chart. The total function calls could not be less than 100, as the data was limited by having been called by at least 100 sites. At around 100 total function calls, the functions have, 1 function call per site on average, and the standard deviation is close to 0. As more data is gathered about the

function, the average and standard deviation may or may not rise, but as less is known then more can be said about how many times that function will be called.



This trend also holds as the threshold is lowered to appearing in 50 sites (as opposed to 100).



Conclusion

The categorical analysis of argument values yields little relevant information in regards to function behavior. There does not seem to be any trends that can separate what arguments will be passed to a function based on its heuristic value. This could be due to the heuristic chosen or the category delimiters, or it could indicate that the specific line of research is fruitless. When examining function calling statistics, however, there seem to be more interesting trends developing.

The correlation analysis indicated that most functions negatively correlated, which is rather unintuitive. It would make more sense that, as the number of function calls a site makes, it keeps calling the same functions to perform the tasks it needs. This would indicate a positive correlation. Instead, the evidence shows that sites that are more active are also a lot more diverse in their script use, and most functions, on average, are not called many times per site.

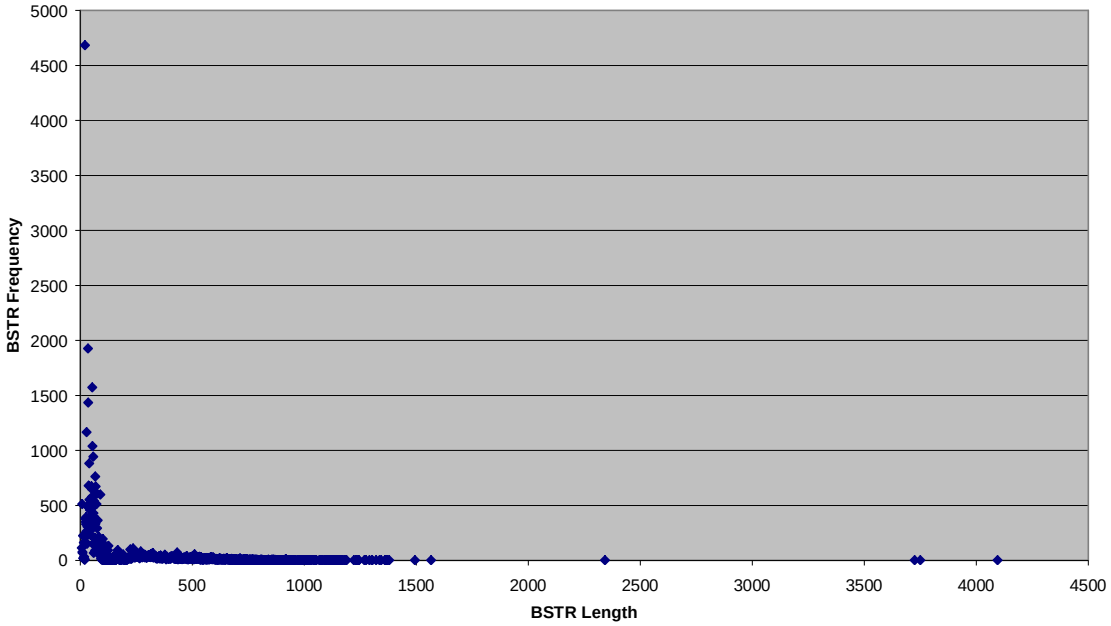
The function calling data also indicates that functions that do not appear in many sites are called less often in sites that they do appear in. This trend in the known data shows that knowledge exists on a function that has not established any trends due to a lack of sufficient information. In essence, it is possible to make an “educated guess” as to whether previously unknown script activity is normal or abnormal with only minimal information about it.

References

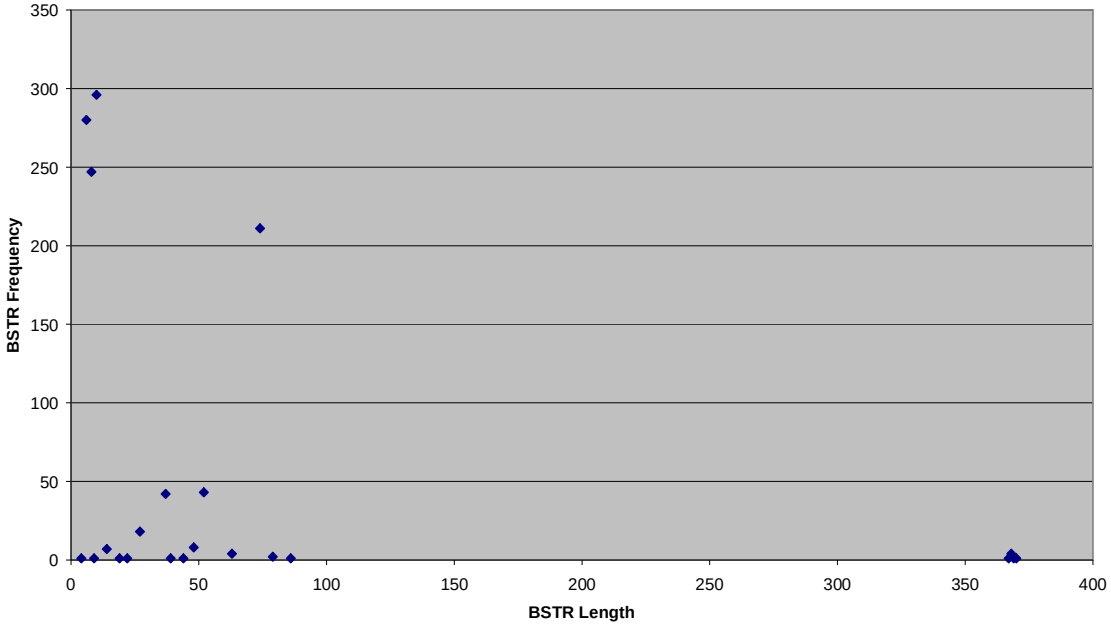
[1] Heritrix – Home Page <http://crawler.archive.org>
[2] 6. Configuring Jobs and Profiles
<http://crawler.archive.org/articles/user_manual/config.html#modules>
[3] IDispatch Interface <http://msdn2.microsoft.com/en-us/library/ms697331(VS.85).aspx>
[4] IDispatch::GetTypeInfo <http://msdn2.microsoft.com/en-us/library/ms221571(VS.85).aspx>
[5] ITypeInfo::GetTypeAttr <http://msdn2.microsoft.com/en-us/library/ms221277(VS.85).aspx>
[6] SourceForge.net: PAMIE <http://sourceforge.net/projects/pamie/>
[7] MSIE Popup Window Address Bar Spoofing Weakness
<http://www.symantec.com/avcenter/attack_sigs/s50029.html>

Appendix A: Distribution as R changes

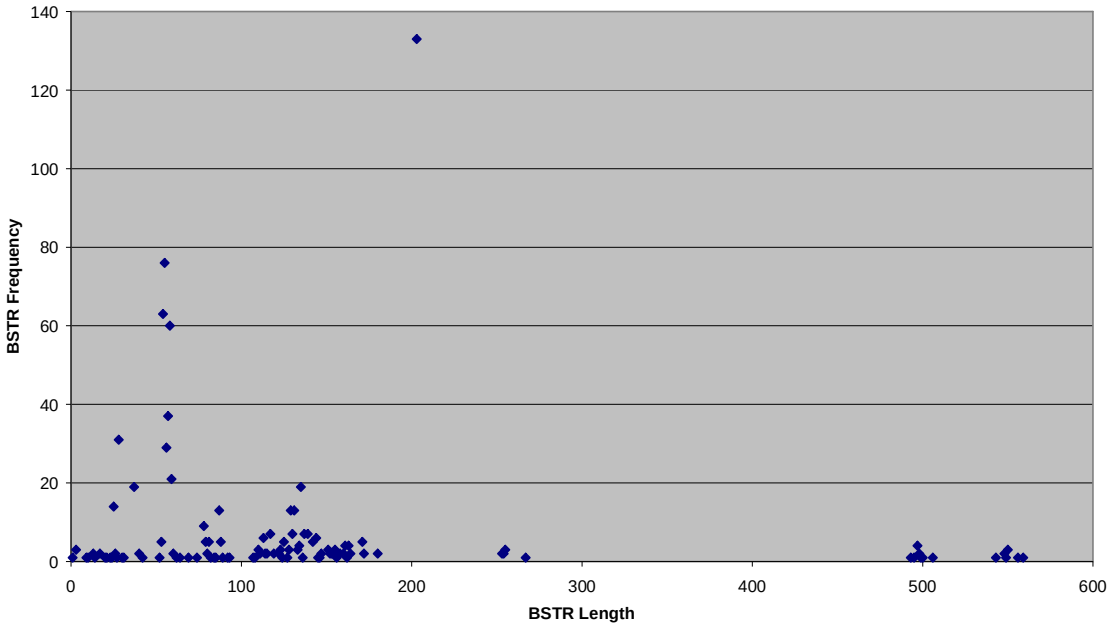
Length/frequency scatter plot for
GUID 3050f51c-98b5-11cf-bb82-00aa00bdce0b, DISPID 1003
(R = 0.092146031)



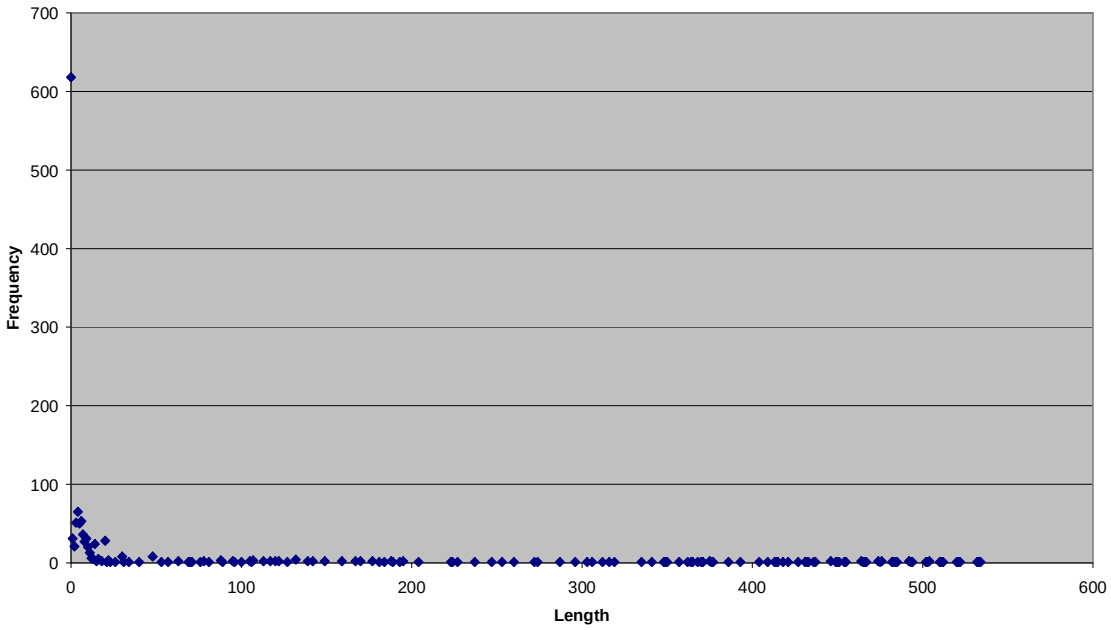
Length/frequency scatter plot for
GUID 3050f524-98b5-11cf-bb82-00aa00bdce0b, DISPID -2147417611
(R = 0.20317546)



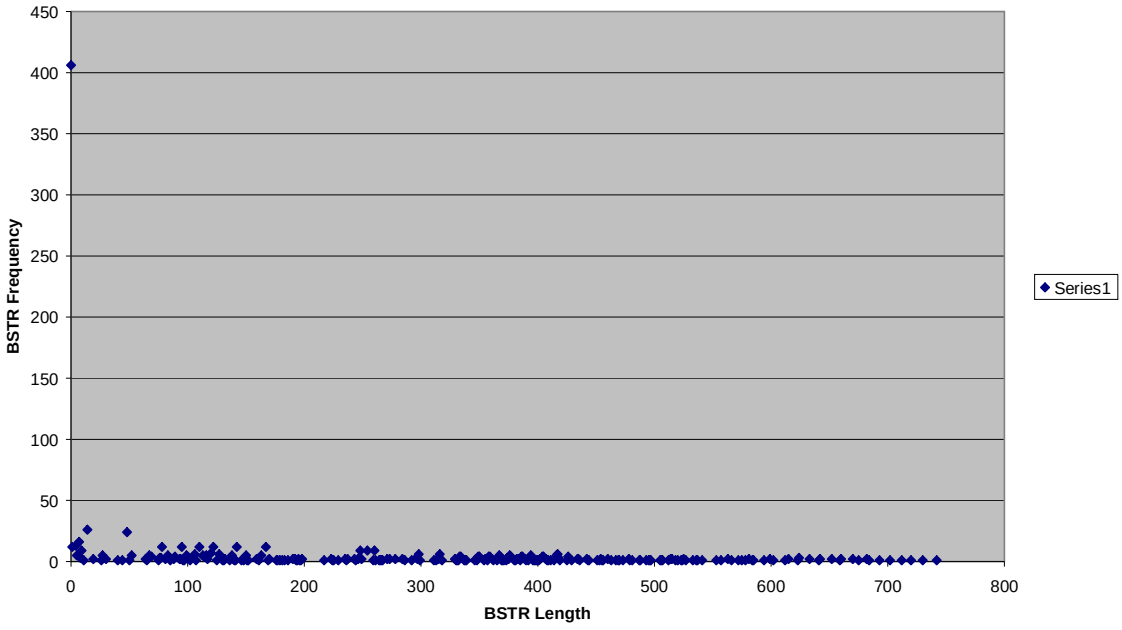
Length/frequency scatter plot for
GUID 163bb1e0-6e00-11cf-837a-48dc04c10000, DISPID 9
(R = 0.328187314)



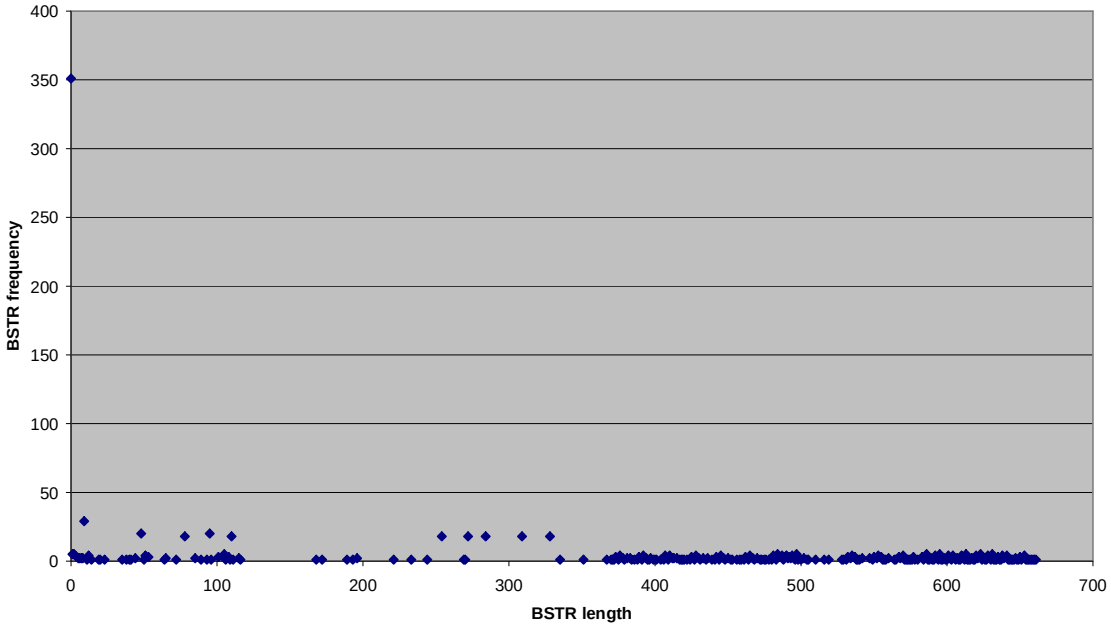
Length/frequency scatter plot for
GUID 3050f55d-98b5-11cf-bb82-00aa00bdce0b, DISPID 3000140
(R = 0.398630311)



Length/frequency scatter plot for
GUID 3050f55d-98b5-11cf-bb82-00aa00bdce0b, DISPID 3000038
(R = 0.498950584)



Length/frequency scatter plot for
GUID 3050f55d-98b5-11cf-bb82-00aa00bdce0b, DISPID 3000061
(R=0.742216663)



Appendix B: Site Statistics

Total number of sites: 14848
Average function calls per site: 5777
Average function calls per function: 1984
Standard deviation of function calls per function: 25493
Standard deviation of function calls per site: 14181

Minus outliers: none
Three Standard Deviations below: 0
Two Standard Deviations below: 0
One Standard Deviation below: 12086
One Standard Deviation above: 1633
Two Standard Deviations above: 510
Three Standard Deviations above: 296
Number of plus outliers according to normal distribution: 323

Median: 1456
First quartile: 438
Third quartile: 4029
Interquartile range: 3591
Minus outliers: none
Lower whisker starts at: 0
Upper whisker ends at: 9365
Number of plus outliers according to “box and whisker” plot: 2048

Total number of functions in “low” category: 4110782
Total number of sites in “low” category: 7566
Total number of functions in “middle” category: 12172758
Total number of sites in “middle” category: 4223
Total number of functions in “high” category: 69487278
Total number of sites in “high” category: 3055
Average number of functions per site in “low” category: 543
Average number of functions per site in “middle” category: 2882
Average number of functions per site in “high” category: 2274