Journal of Theoretical and Applied Information Technology
30th November 2015. Vol.81. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

CORO : GRAPH-BASED AUTOMATIC INTRUSION DETECTION SYSTEM SIGNATURE GENERATOR FOR E-VOTING PROTECTION

[1]SUPENO DJANALI, [2]BASKORO ADI P., [3]HUDAN STUDIAWAN, [4]RADITYO ANGGORO, [5]HENNING T.C
Department of Informatics, Faculty of Information Technology, Institut Teknologi Sepuluh Nopember
Email : [1]supeno@its.ac.id, [2]baskoro@if.its.ac.id, [3]hudan@if.its.ac.id, [4]onggo@if.its.ac.id, [5]henning@if.its.ac.id

ABSTRACT

Attacks on computer network are increasing everyday and most institution use Intrusion Detection System (IDS) to cope with that and most used IDS is the signature-based IDS, which need a database of rules when looking for an malicious packet. Yet there are two problems with this kind of IDS, first, not all people are able to create a signature or rule, therefore they need to wait for updates if they want to renew their database. Secondly, zero-day attack, attack that has never been happened before, is the main weakness of this IDS due to absence of its signature.

We proposed Coro, an IDS signature generator that create an IDS rules based on honeypot log data. Coro uses graph clustering that make it be able to cluster data without the need to recompute the centroid. Coro focuses on HTTP, as it will be used to harden our e-voting system, but it is possible to be extended to other protocols.

Our experiment showed that Coro was able to cluster around 5000 request in a short time and our graph clustering was a big help to that. Moreover, two threshold value used and data preprocessing in that experiment affected amount and quality of the generated rules.
Keyword *: IDS, Rules Generation, Graph Clustering, E-Voting, Graph Mining*

1. INTRODUCTION

Intrusion Detection System (IDS) is something that can be used to protect computer network by alerting administrator when an attack has been occurred. With that alert, administrator can take some actions to prevent the attack from going further. Based on how they detect intrusion, there are two kinds of IDS, Anomaly-based IDS and Signature-based IDS.

Signature-based IDS works by matching incoming/outgoing data with a set of rules/pattern. If any of them are matched with the rules, then it can be concluded that an attack has happened. This method is very effective against well-known attacks, since popular attack string must have some pattern. Most of IDS for production use this, such as Snort, Prelude, and Suricata [1], [2], [3].

Weakness of signature-based IDS is it cannot detect new kind of attack, which usually called Zero-Day Attack. Because at its first appearance nobody know the pattern of the attack, signature-based IDS will not be able to catch them. This type of IDS is heavily depend on the collection of signature/rules, yet making a rule is not a simple thing that everyone can do. Signature-based IDS without rules is useless.

To overcome that weakness, anomaly-based IDS's were made. This kind of IDS works by the principle that behavior of normal users is repetitive action and have some statistical pattern. For example, normal people could forget their password sometimes. But if a system logs wrong passwords were inserted many times, more than usual, it might be an attack. Main method of this type is to find the anomaly from the normal behavior. That enables them to detect unknown attack. Furthermore, human intervention can be limited. The system should be able to learn by themselves. But there is several problem that make this system is hardly implemented [4], [5].

Anomaly-based IDS needs to know what normal behavior looks like, but it is really hard to know what normal behavior is. From the previous example, what if there is a person who is really forgetful. He enters the wrong password more than usual, should it be considered as an attack?

Another limitation of anomaly-based IDS is it usually takes more processing power than the signature-based. It could come from data preprocessing or main statistical computation.

Preprocessing data in anomaly-based IDS is not as simple as in signature-based IDS, which only arrange incoming packet in correct order, but it needs to count some statistical features i.e. mean

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

and variance, and turn the packet into a correct form before inserted into the algorithm. Therefore, signature-based IDS is still preferred today.

In this paper, we develop a system named Coro to automatically generate IDS rules from HTTP logs that was taken from a honeypot. Honeypot has great amount of access log, but it is really disorganized and each data seems unrelated. But actually, each attacker's request is related, for example when they use same tools to attack the server. We sought that every related attack can be concluded as a single rule for IDS, thus make it easier to create a new IDS rule. Furthermore, if the attack was a zero-day attack, then it would be very beneficial for us, since we could know a zero-day attack as soon as possible. In the end, the system will be used for hardening our electronic voting system which is still in progress.

The rest of this paper is structured as follows, section II is talk about related works in signature generation. Our methodologies will be explained in detail in section III. And then we conducted experiments to see the result of our proposed method in section IV. Last but not least, all of these works come into conclusion in section V.

2. RELATED WORKS

Nebula [6] is a signature generator for Snort. It works by utilizing honeypot log data to create a signature and because of that it considers all of the input are malicious. The authors claimed that Nebula was compatible with all honeypots which had previously built, but mostly used with Argos [7] and honeytrap [8].

The signature generator uses a distance graph to cluster similar request. And its strength come from its ability to do real time generator, since Nebula does not need to rebuild the cluster every time a request come, its graph can be recompute in a second. From each cluster, Nebula will create signatures by using generalized suffix tree.

Nebula was released as an open source project and last updated in 2013. From our experiment, Nebula did not work well if the data is too small. Moreover, Nebula failed to generate any signature from requests had came from Glastopf[9] and HoneyD[10].

Honeycomb [11] works like Nebula. It uses honeypot as the source data to create a signature. The main difference between them is the generation algorithm and supported honeypot. Honeycomb only compatible with HoneyD that makes it could get less data than Nebula. Ukkonen's algorithm for finding longest common substring (LCS) is used by honeycomb to search for pattern similarity in a clustered data.

In contrast to Nebula, Honeycomb clusters incoming data based on traffic flow. Every packet comes from same IP address or similar packet comes to same port is clustered. But it did not consider if there are similar packet with same protocol but with different port address. After the clustering process finishes, LCS algorithm will harvest the signature from available clusters, and make it into Pseudo-Snort or BroIDS[12] format.

Unlike Nebula which need a honeypot to be its source of data, Anagram [13], which is a further development of PAYL**Error! Reference source not found.**, is actually an anomaly-based intrusion detection system with an additional feature which is signature generation. N-Gram is used by Anagram to detect malicious packet. In a simple way, anagram compute high order 3-to-9-gram of known benign packet then save them with Bloom Filter to increase space efficiency. Beside computing n-gram from benign packet, Anagram also find n-gram value of Snort signatures and known viruses.

When data come into the system, anagram will compute its n-gram again. Its core hypothesis is any new exploit contains a portion of data that has never been seen anywhere. Thus if any new n-gram, which does not exist in its previous benign n-gram database, then those packet are considered malicious. Besides that, the packet is also checked against known malicious n-gram (created from Snort signatures and known viruses).

Its signature generation feature makes use of detected malicious packet. Those packets have related n-gram then from the n-gram Anagram creates the signature. Figure 1 shows sample signature of an attack generated by Anagram.

Vollmer, et.al. [15] tried to make another autonomous rule creation for IDS and just like Anagram it does not need honeypot as its source. Because the author assumed that the input traffic is already detected as malicious, but it did not mention anything about using honeypot. So it needs separate detection mechanism to distinguish benign and malicious traffic.

Vollmer's method only supports ICMP traffic to be analyzed and uses genetic algorithm to create a signature. Moreover, it only considers ICMP header information, not the payload, which makes it more suitable in an attack which plays with number of traffic such as DDoS, i.e. ICMP flood. Besides, most state of the art attacks do not utilize ICMP anymore; they would rather exploit another protocol weaknesses or application vulnerabilities.

Journal of Theoretical and Applied Information Technology
30th November 2015. Vol.81. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

```
N=3:
*?ph*bb_*//8*p;wg*n;c*n;./c*n;ec*0YYY;echo|H*26.U*1;).*
N=5:
*ums/ad*in/admin_sty*.phpadmin_sty*hp?phpbb_root_path=http://81
.174.26.111/cmd*cmd=cd%20/tmp;wget%20216*09.4/criman;chmod%2074
4%20criman;./criman;echo%20YYY;echo|  HTT*6.26.Use*5.1;)..*
N=7:
*dules/Forums/admin/admin_styles.phpadmin_styles.php?phpbb_root
_path=http://81.174.26.111/cmd.gi*?&cmd=cd%20/tmp;wget%20216.15
.209.4/criman;chmod%20744%20criman;./criman;echo%20YYY;echo|
HTTP/*59.16.26.User-*T 5.1;)..*
```

Figure 1. Sample Signatures Generated By Anagram With Different N Values

3. METHODOLOGIES

Our proposed method works like Nebula and Honeycomb. Coro is specially built for HTTP, considering that it is common used in daily activities and our e-voting system will use it, so a HTTP-based honeypot is needed. We used our own honeypot not Glastopf or other HTTP-based honeypot because our experiment showed that either their log data are hard to be processed later or their system were not too attractive for attacker, thus preventing us from getting more realistic data. A slight improvement has been made to our honeypot and will be written on the following subsection. Next subsections talks about our graph clustering method and the rules generation method respectively.

3.1. Honeypot

Our honeypot are based on the previous work [16] with small modification on what data should be saved. This honeypot works almost like Glastopf, but it was specially designed to attract more human attacker. Their main difference is web pages shown to the users, Glastopf make a web page from a collection of error string to fool hacking tools, while our honeypot use a seemingly dynamic usable web page. It imitates a institution news website, so any human attacker would think that it is a real website, not a honeypot. Our experiment showed that this system could get more data than Glastopf in a same time period.

Somehow, when the experiments were running, we saw that some people used both query string and POST data, regardless to their HTTP method. Thus both query string and POST data must be saved individually, because they have important information. Then we put this honeypot on a public IP address among other existing website in our department for six weeks to collect data. From that experiment we successfully collect about 42,435 requests.

3.2. Graph Clustering

Honeypot log data are various; it could be hard to find a similar string pattern. So we need clustering to gather related request then string patterns will be searched from those resulting clusters. We store the log data in a graph $G = \{ V, E \}$, where V is a set of vertices and E is a set of edges connecting the vertices. Each vertex $v \in V$ denotes an incoming request which has several attributes $v[name, url, queryString, postBody, isRoot, traced]$.

Graph clustering was chosen due to its ability to compute cluster membership of new data without having to recomputed the centroid, thus make the signature generator able to work in real time. Every time a HTTP request come into honeypot and logged, Coro periodically take those requests and make them as vertices V of the graph G. But requests with no query string or POST data and shorter than then character are omitted, because the request URI is too short to be calculated later. Before that, the requests are preprocessed to ensure no unnecessary characters are included.

After the vertex is created, by using Levenshtein algorithm that request is compared to the existing vertices to find how similar that request to the other. Since there are two parameter that we use to compute the distance, query string and POST data, whilst our edge must have single value then Euclidian distance is used. So the formula to find distance between two request can be written as formula (1) :

$$w = \sqrt{Lev(v_N.QS, v_E.QS)^2 + Lev(v_N.PB, v_E.PB)^2} \quad \ldots (1)$$

Where w denotes weight of edge, V_N and V_E is new and existing vertex respectively. QS is Query String attribute of the vertex and PB is Post Body attribute.

From the obtained distance, similarity between two requests can be estimated. Two vertex will be connected if their distance is smaller than

Journal of Theoretical and Applied Information Technology
30th November 2015. Vol.81. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

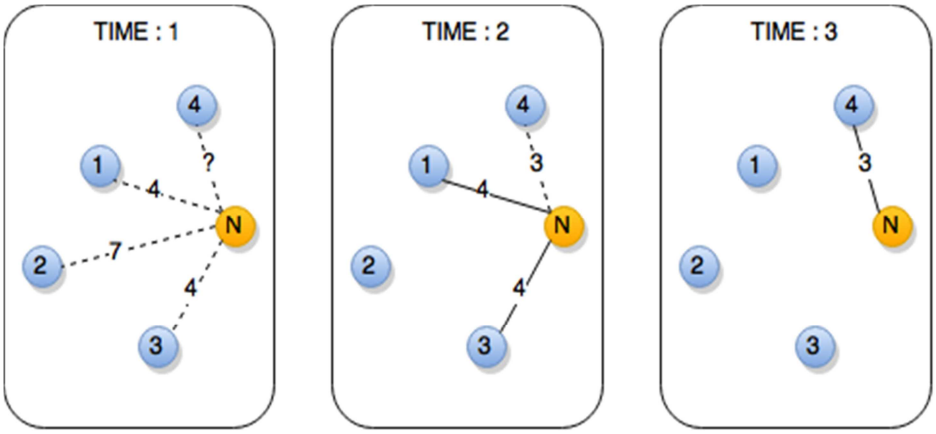

Figure 2. Illustration Of Coro's Graph Clustering

predetermined threshold value T_{EW} or smaller than previous smallest distance of the vertex and the calculated value *weight* becomes weight of edge *e* that connects v_{new} and $v_{existing}$. Let's say there is a new vertex v_N which has distance of 4, 7, 4 to other three existing vertices v_1, v_2, v_3 respectively and T_{EW} for connecting vertices is 5, then fourth existing vertex v_4 has distance of 3 with v_N, therefore existing connection among the v_N, v_1, and v_3 will be broken, and replaced with an edge between the v_N and v_4. Illustration of this story is described on Figure 2.

Resulting graph might consist of more than one fully separated subgraphs $S \in G$. Different from tree, it is hard to determine which node/vertex is the root node of the subgraph, as starting point when traversing the graph. Therefore to find those subgraph, each vertex *v* has attribute *isRoot* with value of true or false. The *isRoot* label is used for determining where the center of the sub graph is. Thus can be used to find how many sub graph are there. Every time a new vertex is connected to another, this label is set to false.

Another attribute given to the vertices is *traced*. This attribute is needed for marking whether the vertex has been visited or not when the traversal process is running. New vertex is always marked as false and Visited vertex is marked as true and will not be traversed again. This can eliminate probability of infinite looping occurrence, since a graph might contain a circular connection.

3.3. IDS Rules Generation

Once requests clustered, the graph *G* is pruned by minimum spanning tree algorithm to avoid too many recursive calls while traversing. Several sub graphs *S* will be analyzed then each sub graph $s \in S$ is to generate as an IDS rule. The process of generating IDS rules from the subgraphs is as followed. First, each root vertex is searched then each subgraph *s* of that root vertex will be traversed. While traversing the graph G, each traversed vertex is marked as 'visited'. Because our algorithm works by visiting a vertex then all of its neighbors, if we do not prevent a visited vertex from being visited again, there will be an infinite loop.

When visiting a vertex, we take either the request URI or POST data as a string. Each string is stored inside an array which will be used to find the longest common substring among those strings. From the longest common substring found, a signature of an HTTP attack is formed and formatted in Snort rules format. Pseudo codes of the graph clustering and rules generation are written on *Figure 3* and *Figure 4*. Note that there is another threshold T_V, which is minimum number of vertices $n = |V|$ in a sub graph *s*. If *n* does not exceed T_V, then there is no need for that sub graph s to be generated as a rule. These steps are repeated in a certain time period. Each newly arrived data will be directly added to the existing graph. Then the rules generation process is repeated for each sub graph.

4. EXPERIMENTAL RESULTS

4.1. Getting Data into Honeypot

Honeypot plays important roles in our system and in order to works well, luring more attacker to the honeypot is important. We placed our honeypot among other public server in our department. We made a default virtual host in our two web servers and used reverse proxy to redirect incoming requests to the honeypot. This was done because we had seen that some attacker tend to scan the server for vulnerable websites or web pages. Few virtual

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

hosts in those servers do not exist anymore, yet the domain is still active and pointed to that servers. Making incoming requests to that domain will be redirected to the default virtual host. Moreover, all of those domains are not in use by neither faculty members nor students that make access to them can be considered as malicious. Architecture of the honeypot and web servers is shown on *Figure 6*.

```
Function AddRequest()
{
	preProcessed(query_string)

	if query_string.length < 10
		return

	new_vertex = Graph.AddVertex(id, URL, query_string, post_data, is_root,
traced)
	foreach vertex in Graph
		if vertex == new_vertex
			continue

		distance = SquareRoot(Lev(vertex.query_string,
new_vertex.query_string)2 +                    Lev(vertex.post_data,
new_vertex.post_data)2)

		if distance <TEW
			threshold = distance
			selected_vertices.add(new_vertex)
			new_vertex.is_root = False
		else if distance == TEW
			selected_vertices.add(new_vertex)
			new_vertex.is_root = False

	foreach vertex in selected_vertices
		Graph.AddEdge(new_vertex, vertex, distance)
}
```

Figure 3. Pseudocode Of Adding A Request To Graph

```
Function TraverseVertex(str_seq, vertex)
{
	foreach v in vertex.neighbors
		if v has not been visited
			str_seq.add(v.query_string)
			v.visited = True
			TraverseVertex(str_seq, v)
}
```

```
Function TraverseGraph()
{
	root_vertices = Graph.FindVertex(is_root = True)

	str_seq = String[][]
	count = 0

	foreach root_vertex in root_vertices:
		if root_vertex has not been visited
			str_seq[count].add(root_vertex.query_string)
			root_vertex.visited = True
			TraverseVertex(str_seq[count], root_vertex)
			if str_seq[count].length > 1 and vertices_in_subgraph > TV
				GenerateRule(str_seq[count])
			count = count + 1
```

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

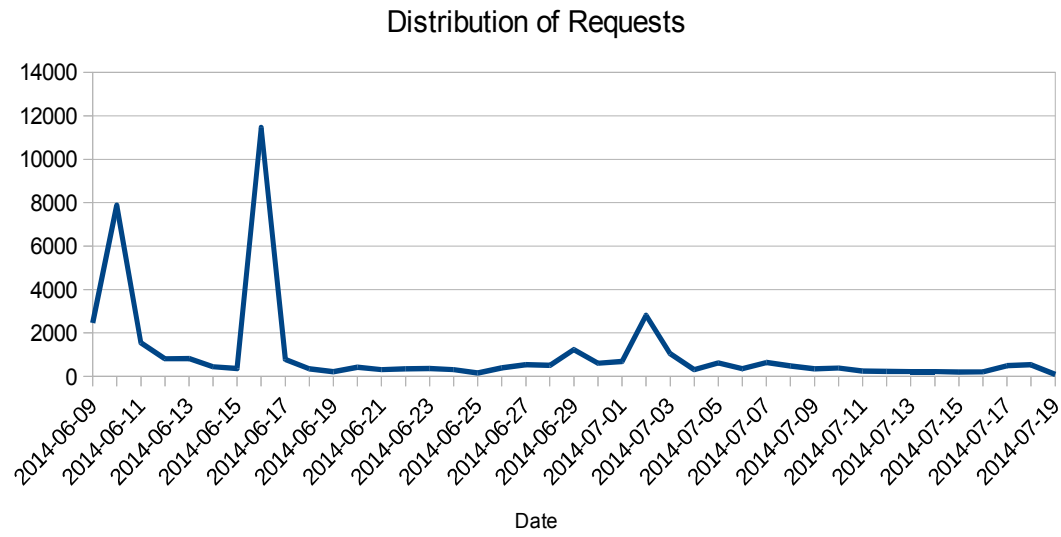

Figure 5. Request Distribution Chart

```
}
```

Figure 4. Pseudocode Of Graph Traversal And Rules Generation

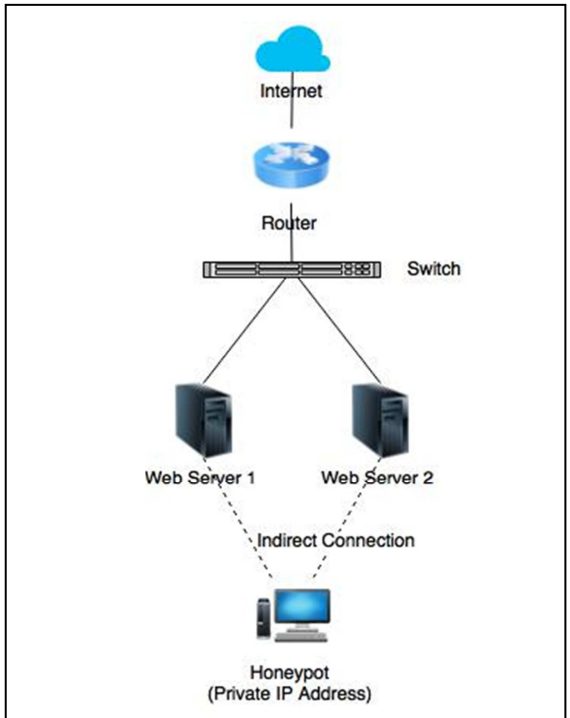

Figure 6. Honeypot Architecture

We run the honeypot for 6 weeks and was able to get 42,435 requests. The data is divided based on the HTTP method, POST, GET, or other method and distribution of requests each day. Statistic of those data is shown on Table 1, while chart of amount of request is shown on Figure 5.

Table 1. Statistic Of Incoming Data

Request Method	Occurrence
GET	35,818
POST	6,432
HEAD	94
OPTION	39
Other	52

4.2. Generating Rules

We only consider requests with any query string or post data because if the request does not have them, the attacker must have used a brute force directory listing attack which is not too important for the rule generator. That's left us with 7,964 from 42,435 requests which denoted as valid requests. It means most of the attacks are directory brute force or looking for sensitive files. But this should be enough for conducting rules generation experiment.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

We tested the program six times with different options. In the first three experiments we used all of 7,964 requests to see effectiveness of our method, but these experiments used different kind of requests. First experiment used raw data in which the requests are still URL encoded string, while the second and third experiment used URL decoded (unquote) string and URL decoded+removing unneeded character (clean), i.e. '/id=', respectively. As shown on Table 2, each experiment resulted different amount of clustered requests due to preprocessing before clustering. Furthermore, amount of sub graphs created were different too, yet the amount of rules created were not too different. If we look at Table 3, rules created by these three experiments are similar, but the preprocessed one produced better and shorter rules. Time needed to cluster and generate rules are shown on , in which just URL decoding can speed up processing time and removing extra characters apparently put additional burden to the system.

In the second three experiments, we did not cluster all of valid requests at once, but they were clustered incrementally to see how Coro handle multiple requests repeatedly. With graph clustering, every time a request comes, we only need to see how far it is with the existing requests. Therefore this should make our proposed method faster than conventional clustering algorithm. This part of experiments is to see that.

From the experiments, we know that total amount of clustered requests, sub graph created, and rules created are exactly same with the previous experiments. But there are three things that can be seen in these experiments. First, A very short rule might be created in the middle of the experiment, but as the requests increasing, it was omitted and replaced with a more specific and longer rule.

Secondly, as shown on Figure 7, time needed to cluster requests, whether it was raw, unquote, or clean, were almost constant until several requests (3500-4000 requests), but after that there was a slight increase. On Figure 9, time needed for rules generation was relatively constant for unquote method, and a bit linear for raw and clean method.

Lastly, just like the all clustered experiments, unquote method always takes less time than the others.

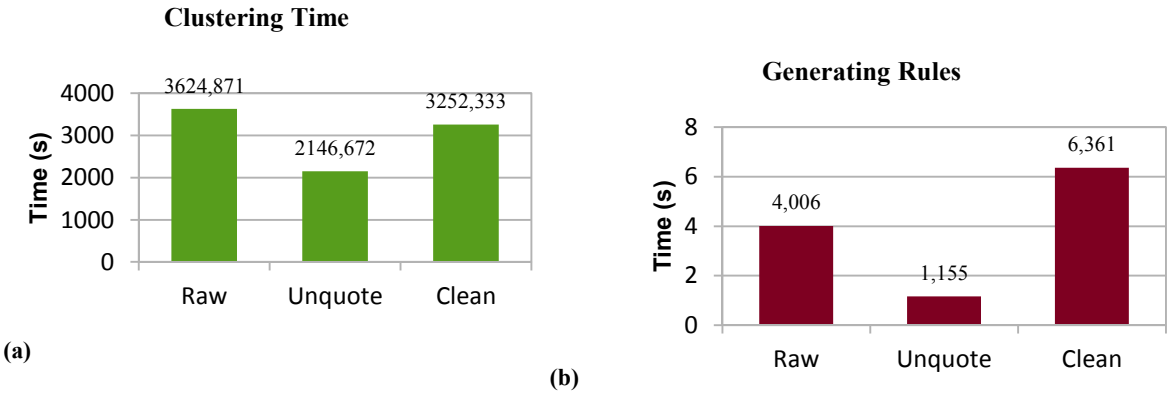

Figure 7. Effect Of Different Preprocessing Method To Run Time

Table 2. Result Of Different Preprocessing

	# of Clustered Requests	# of Sub Graphs	# of Rules Created
Raw	6116	348	11
Unquote	5992	271	9
Clean	5914	290	10

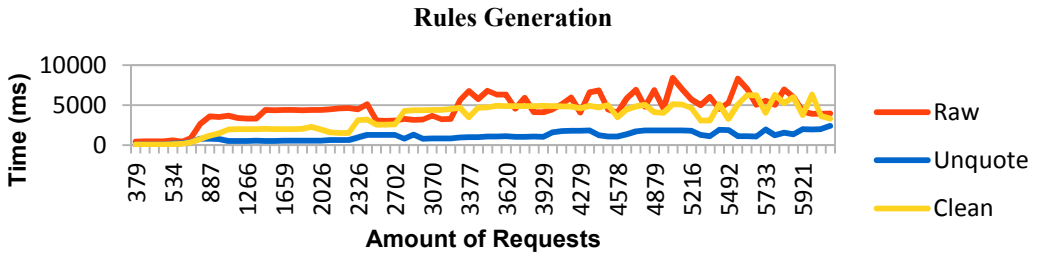

Figure 11. Effect On Rules Generation Running Time In Incremental Experiment

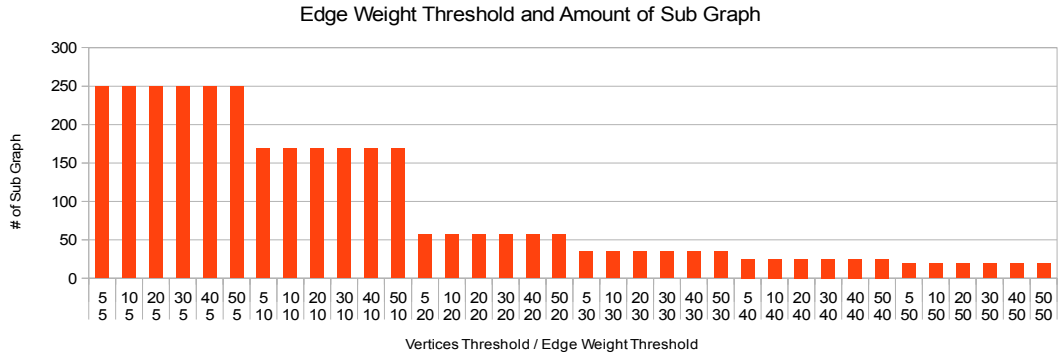

Figure 10. Effect Of Edge Weight Threshold On Amount Of Sub Graph Created

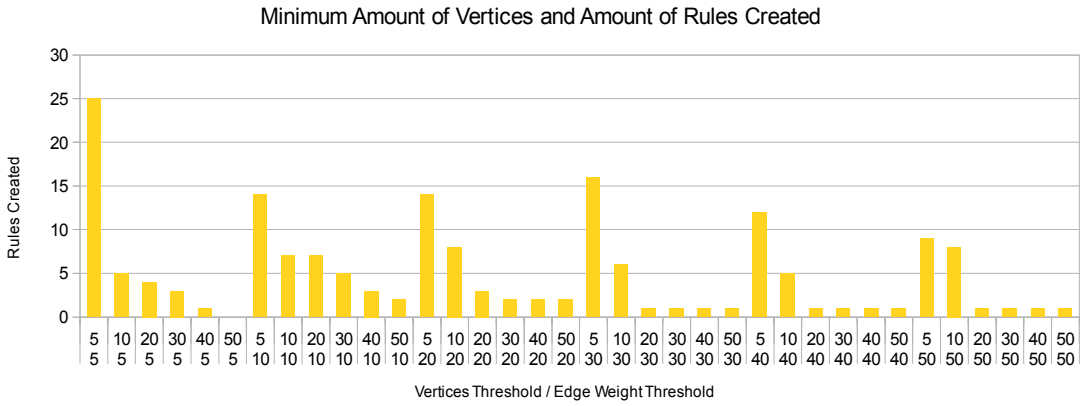

Figure 9. Effect Of Vertices Threshold On Amount Of Rules Created

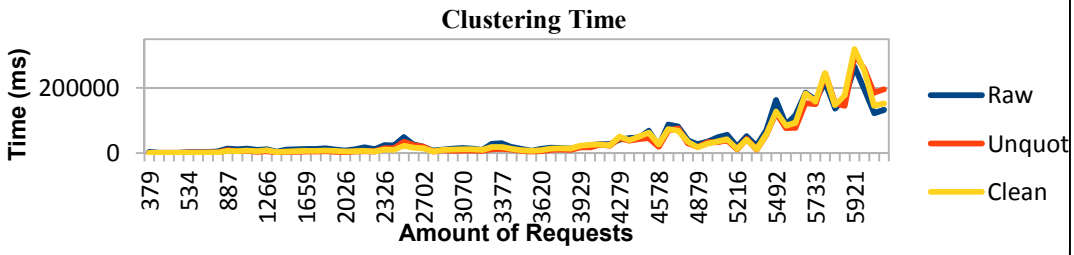

Figure 8. Effect On Clustering Running Time In Incremental Experiment

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Threshold Effect

We took 1000 valid requests and tried to generate rules from them with different threshold value to see its effect. As written above, there are two threshold value used in Coro, threshold for minimum distance between two vertices to be connected (Edge Weight Threshold) and minimum amount of vertices in a sub graph to be computed then (Vertices Threshold). In this part we tried several combinations of those two threshold value and see how they affect the result. We used value of 5, 10, 20, 30, 40, 50 for both of the threshold and unquote method.

The Edge Weight Threshold has the most effect for the amount of sub graph created. It can be seen from the chart on Figure 10 that whatever the Vertices Threshold value, amount of sub graph created were always same and needless to say that for every increment of this value, the amount of sub graph created decreased.

The other value, Vertices Threshold, affected the amount of rules created. As this value getting bigger, the number of rules created was getting smaller. Nevertheless this information cannot be seen as a conclusion yet. We saw more than just a number regarding change of this value, but quality of the rules was also affected.

For example, as shown on Figure 9, some rules were too specific if this value was too small, mostly when we used five. Furthermore, repetitive rules were occurred due to this small value. But if this value was too big, we found that rules created were too general and they included a portion our honeypot specific string, which could mess up detection system. This must have happened because we used unquote method. From our experiment, we think that the value of ten is the best for Vertices Threshold. With that value, repetitive rules could be eliminated and rules created were not too specific nor too general. Examples of those rules can be seen on Table 4 and Table 5 respectively.

5. CONCLUSION AND FUTURE WORKS

Coro was successfully built as IDS signature generator and the result of our experiment shows that Coro is able to work incrementally as the data come. Though created rules still depend on the threshold values and need human evaluation. There are some challenges left in this topic, such as speeding up the clustering computation when the data is so large and more filtering to the rules created, so repetitive, too specific, or too general rules can be eliminated. Hopefully we can expand Coro to be able to compute not only HTTP traffic but other protocols as well.

Table 3. Example Of Generated Rules Based On Preprocessing Method

Method	Rules Content
Raw	alert tcp any any -> any 80 (content: "%29%20and%20"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "select%2"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "%270%3a0%3a"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "%20from%20pg_sleep%28"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "%3ddbms_pipe.receive_message%28chr%28"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "%26highlight%3d%2527.passthru%28%24http_get_vars%5brush%5d%29.%2527"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "-d%2ballow_url_include%3don%2b-d%2bsafe_mode%3doff%2b-d%2bsuhosin.simulation%3don%2b-d%2bdisable_functions%3d%22%22%2b-d%2bopen_basedir%3dnone%2b-d%2bauto_prepend_file%3dphp%3a//input%2b-d%2bcgi.force_redirect%3d0%2b-d%2bcgi.redirect_status_env%3d0%2b-d%2bauto_prepend_file%3dphp%3a//input%2b-n"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "-d%2ballow_url_include%3don%2b-d%2bsafe_mode%3doff%2b-d%2bsuhosin.simulation%3don%2b-d%2bdisable_functions%3d%22%22%2b-d%2bopen_basedir%3dnone%2b-d%2bauto_prepend_file%3dphp%3a//input%2b-d%2bcgi.force_redirect%3d0%2b-d%2bcgi.redirect_status_env%3d0%2b-n"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "%27%22--

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Method	Rules Content
	%3e%3c/style%3e%3c/script%3e%3cscript%3enetsparker%280x0000"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "netsparker"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "id%3d2%27%20union%20all%20select%201%2cemail_kontributor%20%2c3%20from%20db_artikel.tb_kontributor%20limit%200%2c1%20--%2b"; nocase; http_raw_uri;)
Unquote	alert tcp any any -> any 80 (content: " from pg_sleep("; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "=dbms_pipe.receive_message(chr("; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: " union all select "; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "&highlight=%27.passthru($http_get_vars[rush]).%27"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "-d+allow_url_include=on+-d+safe_mode=off+-d+suhosin.simulation=on+-d+disable_functions=""+-d+open_basedir=none+-d+auto_prepend_file=php://input+-d+cgi.force_redirect=0+-d+cgi.redirect_status_env=0+-d+auto_prepend_file=php://input+-n"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "-d+allow_url_include=on+-d+safe_mode=off+-d+suhosin.simulation=on+-d+disable_functions=""+-d+open_basedir=none+-d+auto_prepend_file=php://input+-d+cgi.force_redirect=0+-d+cgi.redirect_status_env=0+-n"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "'"--><\/style><\/script><script>netsparker(0x0000"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "netsparker"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "id=2' union all select 1,email_kontributor ,3 from db_artikel.tb_kontributor limit 0,1 --+"; nocase; http_raw_uri;)
Clean	alert tcp any any -> any 80 (content: " from pg_sleep("; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "=dbms_pipe.receive_message(chr("; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: " union all select "; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "71,floor(rand(0)*2))x from information_schema.character_sets group by x)a)"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: ") then 1 else 0 end))::text||(chr(113)||chr("; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "&highlight=%27.passthru($http_get_vars[rush]).%27"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "-d+allow_url_include=on+-d+safe_mode=off+-d+suhosin.simulation=on+-d+disable_functions=""+-d+open_basedir=none+-d+auto_prepend_file=php://input+-d+cgi.force_redirect=0+-d+cgi.redirect_status_env=0+-d+auto_prepend_file=php://input+-n"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "-d+allow_url_include=on+-d+safe_mode=off+-d+suhosin.simulation=on+-d+disable_functions=""+-d+open_basedir=none+-d+auto_prepend_file=php://input+-d+cgi.force_redirect=0+-d+cgi.redirect_status_env=0+-n"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: ""--

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Method	Rules Content
	></style></script><script>netsparker(0x0000"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "' union all select 1,email_kontributor ,3 from db_artikel.tb_kontributor limit 0,1 --+"; nocase; http_raw_uri;)

Table 4. Rules Created Due To Wrong Threshold Value

Category	Sample of Rules Created
Too Specific Rules	alert tcp any any -> any 80 (content: " and (select 1259 from(select count(*),concat(0x7174686871,(select (case when (1259=1259) then 1 else 0 end)),0x7161717671,floor(rand(0)*2))x from information_schema.character_sets group by x)a)"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: " and 2486=cast((chr(113)||chr(116)||chr(104)||chr(104)||chr(113))||(select (case when (2486=2486) then 1 else 0 end))::text||(chr(113)||chr(97)||chr(113)||chr(118)||chr(113)) as numeric)"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: " and 6391=convert(int,(select char(113)+char(116)+char(104)+char(104)+char(113)+(select (case when (6391=6391) then char(49) else char(48) end))+char(113)+char(97)+char(113)+char(118)+char(113)))"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: " and 9676=(select upper(xmltype(chr(60)||chr(58)||chr(113)||chr(116)||chr(104)||chr(104)||chr(113)||(select (case when (9676=9676) then 1 else 0 end) from dual)||chr(113)||chr(97)||chr(113)||chr(118)||chr(113)||chr(62))) from dual)"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "id=<iframe src=""; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: "' union all select null,(select concat(0x7174686871,column_name,0x716769676573,column_type,0x7161717671) from information_schema.columns where table_name=0x74625f6"; nocase; http_raw_uri;)
Repetitive Rules	alert tcp any any -> any 80 (content: " limit 1,1 union all select null#"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: " limit 1,1 union all select null, null#"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: " limit 1,1 union all select null, null, null#"; nocase; http_raw_uri;) alert tcp any any -> any 80 (content: " limit 1,1 union all select null, null, null, null#"; nocase; http_raw_uri;)
Too General Rules	alert tcp any any -> any 80 (content: "id="; nocase; http_raw_uri;)

Table 5. Rules Created With Edge Weight Threshold 5 And Vertices Threshold 10

```
alert tcp any any -> any 80 (content: "; select "; nocase; http_raw_uri;)
alert tcp any any -> any 80 (content: " waitfor delay '0:0:5'"; nocase;
http_raw_uri;)
alert tcp any any -> any 80 (content: " order by "; nocase; http_raw_uri;)
alert tcp any any -> any 80 (content: "id="; nocase; http_raw_uri;)
alert tcp any any -> any 80 (content: " union all select null"; nocase;
http_raw_uri;)
```

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

REFERENCES

[1] Roesch, M. (1999, November). Snort: Lightweight Intrusion Detection for Networks. In LISA (Vol. 99, No. 1, pp. 229-238).

[2] Zaraska, K. (2003). Prelude IDS: current state and development perspectives. URL http://www. prelude-ids. org/download/misc/pingwinaria/2003/paper. pdf.

[3] Suricata. (2015). Homepage: http://suricata-ids.org/

[4] Denatious, D. K., & John, A. (2012, January). Survey on data mining techniques to enhance intrusion detection. In Computer Communication and Informatics (ICCCI), 2012 International Conference on (pp. 1-5). IEEE.

[5] Sabahi, F., & Movaghar, A. (2008, October). Intrusion detection: A survey. In Systems and Networks Communications, 2008. ICSNC'08. 3rd International Conference on (pp. 23-26). IEEE.

[6] Werner, T., Fuchs, C., Gerhards-Padilla, E., & Martini, P. (2009, October). Nebula-generating syntactical network intrusion signatures. In Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on (pp. 31-38). IEEE.

[7] Portokalidis, G., Slowinska, A., & Bos, H. (2006, April). Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In ACM SIGOPS Operating Systems Review (Vol. 40, No. 4, pp. 15-27). ACM.

[8] Yasinsac, A., & Manzano, Y. (2002, July). Honeytraps, a network forensic tool. In Sixth Multi-Conference on Systemics, Cybernetics and Informatics.

[9] Rist, L., Vetsch, S., Koßin, M., & Mauer, M. Know your tools: Glastopf—A dynamic, low-interaction Web application honeypot. 2011.

[10] Provos, N. (2003, February). Honeyd-a virtual honeypot daemon. In 10th DFN-CERT Workshop, Hamburg, Germany (Vol. 2, p. 4).

[11] Kreibich, C., & Crowcroft, J. (2004). Honeycomb: creating intrusion detection signatures using honeypots. ACM SIGCOMM Computer Communication Review, 34(1), 51-56.

[12] Bro, I. D. S. (2008). Homepage: http://www. bro-ids. org.

[13] Wang, K., Parekh, J. J., & Stolfo, S. J. (2006, January). Anagram: A content anomaly detector resistant to mimicry attack. In Recent Advances in Intrusion Detection (pp. 226-248). Springer Berlin Heidelberg.

[14] Wang, K., & Stolfo, S. J. (2004, January). Anomalous payload-based network intrusion detection. In Recent Advances in Intrusion Detection (pp. 203-222). Springer Berlin Heidelberg.

[15] Vollmer, T., Alves-Foss, J., & Manic, M. (2011, April). Autonomous rule creation for intrusion detection. In Computational Intelligence in Cyber Security (CICS), 2011 IEEE Symposium on (pp. 1-8). IEEE.

[16] Djanali, S., Arunanto, F. X., Pratomo, B. A., Baihaqi, A., Studiawan, H., & Shiddiqi, A. M. (2014, November). Aggressive web application honeypot for exposing attacker's identity. In Information Technology, Computer and Electrical Engineering (ICITACEE), 2014 1st International Conference on (pp. 212-216). IEEE.