UNIVERSITY OF AMSTERDAM

The use of workflow topology observables in a Security Autonomous Response Network

Master of Science Research Project

Adriaan Dens
adriaan.dens@os3.nl

Supervisors: Prof. dr. R.J. Meijer and Ir. M.X. Makkes

July 25, 2015

Abstract

With the trend of ever increasing data, applications and networks, there is a need for autonomous control by the network. In this thesis, we look at workflow topologies which process data and how these topologies can be dynamically monitored and adapted by a control loop. This control loop can be used to monitor for - and respond to security incidents, thereby implementing a security autonomous response network. This thesis presents a solution of estimating the observables of workflow topologies and comparing them to the actual topology, as an implementation of the control loop. To test the estimations, a simulator was developed as a proof of concept. This proof of concept shows good results for the estimations once the topology has become a free-flow network. However, more testing should be done to show that these estimations perform equally well on a physical network.

Contents

1 Introduction

In the last few years, a development towards programmable networks has been seen [1]. Programmable networks allow network engineers to completely control their networks in software. This allows, for instance, the changing of network routes using application programming interfaces (APIs) of the network devices rather than having to manually configure these devices.

Applications, possibly forming a complex distributed application, are deployed on top of this software controlled infrastructure. This thesis looks at distributed applications, which form workflow topologies. An example of a workflow topology is shown in Figure 1.

Because of the programmability of the network, a software control loop can be written to dynamically alter the behaviour and performance of the workflow topology. For example, the control loop can perform scaling up actions for the application if an increase in traffic is seen.

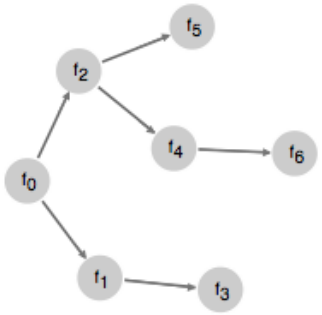


Figure 1: *Logical workflow topologies consist out of functions and relations between those functions. Changes in this topology are the result of the control loop.*

An example of the scaling of a workflow topology can be seen in Figure 2, which is a scaled out version of the logical topology of Figure 1. The control loop can even become autonomous, resolving issues without human interaction. This concept of autonomous control is becoming more and more relevant because of the increase in the size of applications and the amount of data flowing through.

Marc Makkes [1] from the SNE[2] Research Group at the University of Amsterdam recently discovered that patterns are generated by the synergy between the control loop, workflow topology and data flowing through.[2] To assess the potential of these patterns, this thesis will investigate if (and how) these can be used for anomaly detection. For example, changes in the observables, resulting in an unknown pattern, can indicate a cyber-attack against the workflow topology.

[1] `https://staff.fnwi.uva.nl/m.x.makkes`
[2] System and Network Engineering

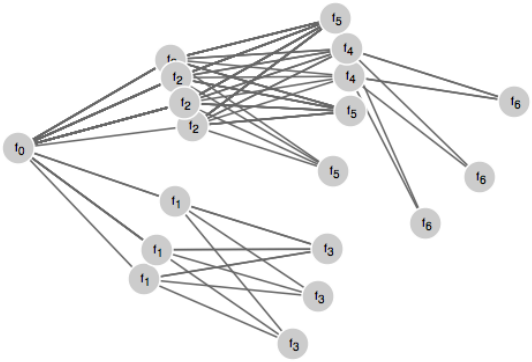


Figure 2: *Shows how the logical topology of Figure 1 has converged under the influence of the control loop and data.*

1.1 Research Questions

How can observables of software controlled workflow topologies be used in Security Autonomous Response networks?

For instance, how can changes in the workflow topology be used to detect security issues and how can one respond automatically to these issues.

Related to the research question, the following sub questions are posed:

- What are the characteristic observables as a result of control loops modifying the workflow?
- What influences the sensitivity of the observables?
- What kind of attacks can be detected by observing changes in workflow observables and what countermeasures can be taken?

1.2 Related Work

Auto-scaling the workflow topology is part of this research project. Lorido-Botrán et al. [3] describe five major techniques to perform auto scaling of resources. Namely threshold rules, Q-learning, queuing theory, control theory and time-series analysis. During this project, the threshold rules algorithm was used to decide whether or not functions in the topology should be scaled. This algorithm was chosen for its simplicity and ease of implementation.

Kaur et al. [4] describe the use of *mininet*[3] as a framework to create Software Defined Networks for research purposes. As described in Section 3.2, our experience with *mininet* for the creation of a workflow topology was less successful. However, it showed essential insights that were used in the proof of concept.

Dimitrov [5] describes the notion of Security Autonomous Response networks as a "network that adjusts itself in order [to] take care of security threats". How a network

[3] `http://www.mininet.org`

can autonomously defend itself against attacks is part of this thesis project.

1.3 Scope

To our knowledge, the subject of workflow topology observables and their relation to the control loop and data has not yet been investigated. As such, it is necessary to begin with the basics and explore this field step by step. Therefore, the research only looked into three workflow topology observables. Namely, the topology itself, the node activity and the link load. Although the original goal of the project was to investigate patterns of the workflow topology and use them in a security context, it was soon realised that a step should be taken back and that a look should be taken into the nodes with functions and the links between functions. Finding discrepancies at those two observables might allow for a more sophisticated correlation at the topology level and as such find deviations in the pattern of a workflow topology. Due to the short period in which this investigation was performed, no investigation was done into the correlation of the discrepancies, found at the node and link level, at the topology level.

In Section 3, machine learning is mentioned as a solution to learn characteristics of the topologies. However, no time was allocated in this project to research which algorithms could be used.

2 Background

This section explains essential concepts that are used in this thesis. Starting with the concept of control loops, followed by an overview of workflow topologies, its observables and scaling, and finishing with security autonomous response networks.

2.1 Control loops

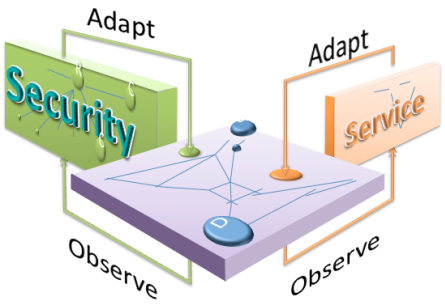


Figure 3: *A security control loop and a service control loop observe and adapt the network [Image: Robert Meijer].*

As mentioned in the introduction section, programmable networks give the opportunity to dynamically alter the network. To continuously monitor and adapt workflow topologies, the concept of a control loop is introduced. This control loop monitors the topology and adapts the topology to handle the current situation. Given congestion in a node, it can decide to deploy a node with similar functionality and update the topology to incorporate this new node.

The control loop can consist of multiple modules, each performing monitoring. Figure 3 shows a network with two such control loops; one is responsible for adapting the network, such as scaling up or down, and the other is responsible for responding to security issues, such as detecting different patterns of the topology.

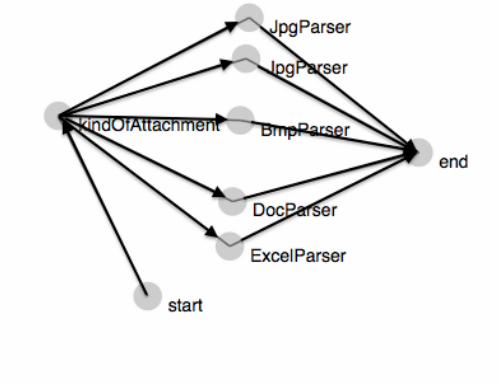


Figure 4: *An example workflow topology with a scaled up JPG parser.*

2.2 Workflow topologies

Workflow topologies consist of functions f_0 to f_n and links, which form relations between certain functions. Workflows consist of creating a flow through these functions using the links. Figure 4 shows an example topology that parses email attachments. The *kindOfAttachment* function determines which parser is selected and thus which function is subsequently selected. The different parsers each signify a parser for a certain type of file.

The goal of workflow topologies is to process data. Data is split up into data items, which are logical units of data. An example is a Twitter[4] stream where each tweet is a coherent data item. Furthermore, in this thesis, an assumption is made that there is a finite number of distinct data types under which the data items can be categorized. Figure 5 shows the resulting workflow generated by a JPG data item.

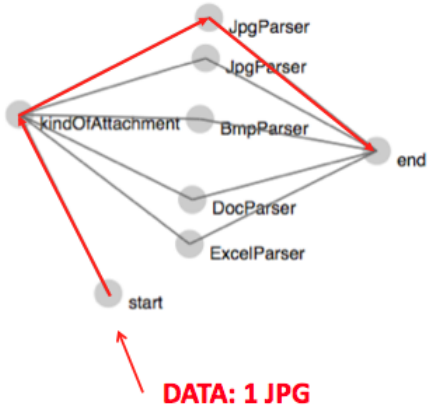


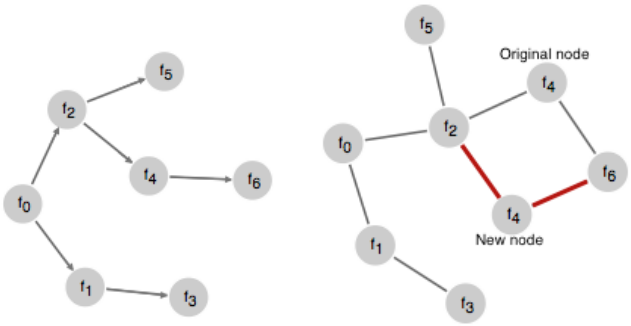
Figure 5: *Sending a JPG data item through the topology creates a characteristic workflow.*

2.3 Scaling of the workflow topology

The influx of data might be more than the topology can handle, resulting in the queuing of data items and the overloading of nodes. To resolve these issues and create a topology where data items can be immediately processed, a so-called free-flow network, the topology has to be scalable. When a node is overloaded with work from data items that go through the function, a scale up can be done by the control loop. This can either be the allocation of more resources or the creation of a new node with the same function such that data items get spread out over multiple nodes. Similarly, a scale down can be done when the node has no processing to do.

In this thesis, workflow topologies are scaled by adding or removing nodes. When a node in a workflow topology has to be scaled up, a new node is created with the same function as the original node. Then, the links of the original node are copied to the new node. An example is shown in Figure 6b, where the relation of f_2 to f_4 and f_4 to f_6 are copied. When a node is scaled down, these relations are removed and the node is destroyed.

[4] `https://www.twitter.com`



(a) *The logical topology.*

(b) *The node with function f_4 is scaled up and relations are copied.*

Figure 6

2.4 Observables of workflow topologies

Observables of workflow topologies are properties of the topology that can be observed by the control loop. The topology itself is an observable of a workflow topology; when data flows through the network, the topology changes in such a way that is characteristic for the data that is flowing through. Namely, every type of data generates workflows on the topology that are characteristic for that type of data, giving the topology a pattern that is related to the data, which flowed through. This change can be observed by the control loop. Furthermore, we can say that the data generates fingerprints of the topology. These fingerprints can be stored and used as a whitelist of possible topologies.

Because workflow topologies might consist out of thousands of nodes with hundreds of functions, an abstract way of looking at a topology is needed. This can be done by looking at the relations between the functions. For example, in Figure 6a, f_4 expects data from f_2 and sends data to f_6. These relations can also be observed by the control loop. In a security context, connections between nodes that do not originally have a relation can indicate a compromised node trying to exploit other nodes in the topology.

Since a topology consists of nodes with functions and links which properties change when data flows through the topology, these are observables of the workflow topology as well.

2.4.1 Node activity observable

When workflows pass through nodes, they increase the activity on those nodes. A data item that hits a node might spawn a new process, allocate some extra memory and require some processing time on the CPU (Central Processing Unit). These are called node activity parameters. Furthermore, one might also look at the changes that a data item brings to files, (memory) page writes, etc. However, this information must be exposed to the control loop. This can be done on the application level or by using standard monitoring tools like SNMP (Simple Network Management Protocol) making the application agnostic to the control loop. This thesis will only look at the CPU load as the node activity parameter.

Discrepancies between the observed node activity and the estimated activity can indicate problems with the node. For example, there might be data items that perform bad on the implementation of the function resulting in a discrepancy. The control loop can pick up such discrepancies. In a security context, these discrepancies can indicate a compromised node.

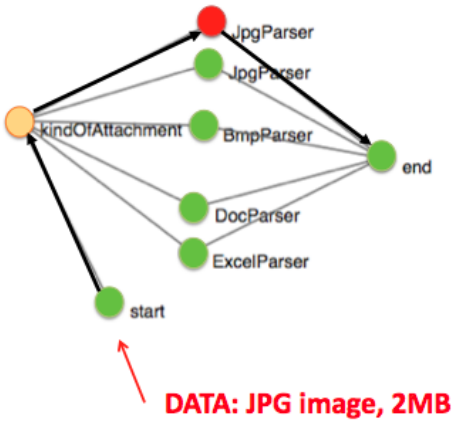


Figure 7: *A JPG data item generates a certain load on the nodes of the functions it hits.*

2.4.2 Link load observable

When data items are sent through links, they increase the load on those links. These links can be physical connections between servers or logical links between two nodes which run on virtual machines, possibly in different datacenters. Depending on the situation, the link load can be retrieved from network equipment such as switches and routers or on the nodes itself, in the case of tunnels. In this thesis, an assumption is made that these connections are dedicated to the workflow topology and that other network traffic, such as ARP (Address Resolution Protocol), is negligible in load or takes up a fixed amount of load and thus can be subtracted from the link load to get the load generated by the data items.

As with the node activity observable, discrepancies in the observed load and estimated load can indicate other usage of the link or a so-called amplification attack on the topology. Amplification attacks are described by the US-CERT [6] as "a single packet [which] can generate tens or hundreds of times the bandwidth in its response". In a workflow topology this is a data item which generates an amplification of the data which is greater than expected.

2.5 Security autonomous response networks

Software Defined Networks and Network Function Virtualisation [7] allow one to control and adapt the network in software. As such, it is possible to dynamically change the network when events occur. Security autonomous response networks take this one step further; not only is the network controlled by software but when security events occur, the network resolves them without human interaction. The network has thus become autonomous. This behaviour is implemented in the control loop as explained in Section 2.1. The control loop is used to detect deviations from the topology which are expected

from the data that resides in the topology.

When given a security violation, the network can respond in a number of different ways. Further investigation of the violation can be done by more closely monitoring the node or link. The network can decide to change the flow through the network so all data flows through monitor nodes such as intrusion detection systems. Sandboxing the node is another option.

When a node in the workflow topology is compromised, the network can respond by removing the node from the topology and creating a node with the same function as the compromised node. Less drastically, the network can reprovision the node, using an orchestration tool such as Puppet[5]. The node can also be converted into a honeypot; the network sends fake data through the compromised node and monitors how the node responds.

Unusual bandwidth usage on the links in the topology can be resolved by altering the flows in the flow table of the Software Defined Networking equipment, explicitly disallowing (blacklisting) the traffic responsible for the high load or only allowing known traffic (whitelisting).

Which response is chosen by the control loop depends on the severity of the issue and the cost to fix the issue. Tearing down part of network, resulting in a partial downtime of the topology is more expensive than sandboxing the compromised node. A chain of decisions can also be taken; for example, sandboxing the node and distributing all data items to other nodes, after which the malicious node is taken down.

3 Methodology

In this section, the approach taken to use workflow topology observables to detect possible security issues with the topology, is explained. The proof of concept used to test the assumptions, is also described.

In order to detect anomalies in the observables, the control loop needs to know the difference between normal and abnormal behaviour. If the control loop has no information about the topology, then the control loop cannot distinguish good from bad behaviour. This implies a learning phase of the control loop in which it learns the difference.

This learning phase can be implemented as a machine learning algorithm or as a manual benchmarking/observation phase. The goal of this phase is to learn the characteristics of the different types of data that can go through the topology. These characteristics are then used to estimate the observables. An example of such a characteristic is the fraction of data items of a given data type that hits a function.

[5] `https://puppetlabs.com/`

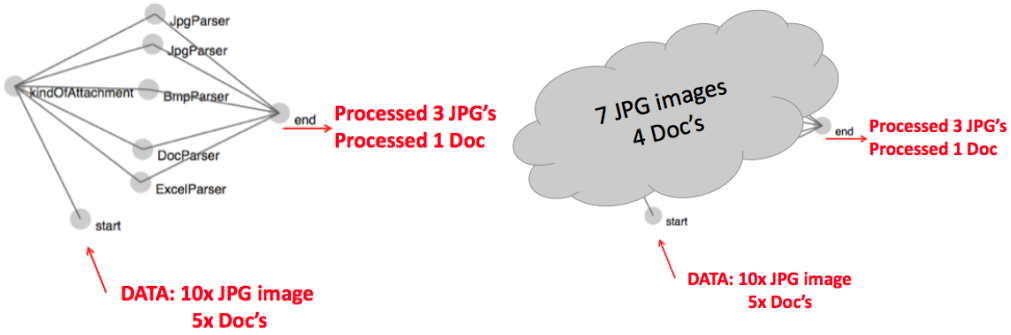


Figure 8: *Knowing which data items are inside the topology can be calculated by taking the difference of the data items which have gone into the topology for processing and the data items that have finished processing.*

Knowing which data items are inside the workflow topology is an important part of the estimation. For this reason, controller nodes are placed in the topology that are used to log where the data has passed. As the name suggests, controller nodes are nodes in the topology that are controlled by the control loop and perform functions on behalf of the control loop. Dedicated controller nodes are chosen above altering the distributed application itself with extra logging capabilities to make the topology as agnostic as possible for the control loop. The *start* and *end* function of Figure 8 are two examples of such controller nodes. These controller nodes can also be implemented as tap devices that get their data from passive sniffing on the network.

If two controller nodes are placed respectively before the data items reach the topology and one after the topology then a comparison can be made between both sets to get the data items which are still being processed by the workflow topology. This initial research does not take into account functions that multiply or demultiply data items; if five data items are inserted in the topology for processing, then five data items are expected to arrive at the end of the topology.

Equation 1 shows this law of conservation.

$$I = S - E \tag{1}$$

where S is the set of data items which entered the topology and E is the set of data items which have finished processing. I thus contains everything which is being processed.

Because the performance of the control loop might suffer given a big topology with many data items, one can choose to place intermediate controller nodes. These controller nodes can be placed in the topology such that calculations can be done on a subset of the data items in the network, thereby decreasing the amount of work each controller node has to do. Placing intermediate controller nodes also allows for more accurately estimating the location of the data items in the topology.

3.1 Estimating the observables

Knowing certain characteristics of types of data, estimations can be made of the observables. These can then be compared to the actual values observed. Large differences between the estimation and the observed value can indicate a problem in the network. To differentiate between abnormalities and normal behaviour, an interval is made around the estimated load. Observed loads that fall outside of this interval are considered to be discrepancies. The interval is defined as $[e - \alpha, e + \alpha]$ where α is the discrepancy limit. If the estimated load is, for example, 50% and the discrepancy limit α is 20% then observed loads between 30% and 70% are not considered to be a problem for the network.

An estimation was made for the node activity, link load and topology observables which were discussed in Section 3.

3.1.1 Node activity parameter estimation

To explain how the estimation was derived, the simple case of the topology in Figure 12 with the data items shown in Figure 8 are used as an example.

As discussed in the background (Section 2.2), data items follow a certain path through the topology. This workflow generates activity on the nodes which it hits. When all data items residing in the topology are known, an estimation of the activity on a function can be made by taking the data items that reside in the network and go through that function and multiply it by the load these data items generate on the node. More generally put, a fraction of certain data item types go through certain functions. This fraction is given by the learning phase. For example, the JPG data item type never (0:1) goes through the document parser but always (1:1) goes through the JPG parser.

In the previous paragraph, the assumption was made that a data item goes through a function. However, data items go through nodes with functions, not through functions itself. As such, the estimation only holds when there is only one node with that function. If there are multiple nodes with a function, only a fraction of the data goes through the node for which the estimation is made. This depends on the load balancing technique used in the workflow topology. Figure 12 shows two JPG parsers, if the assumption is made that the application uses round-robin load balancing, then one in two data items go through the node for which the estimated load is calculated.

Until now, another assumption was made. Namely, we assumed that when observing the current activity in a node, all data items that can be processed by a node, are currently being processed. This might not be the case, however, certain data items might be in another node when we observe the node activity. This aspect of time has to be taken into account. As shown in Figure 9, given a chain of functions where one function takes up 90 percent of the time; when a sample is made, a fraction of nine out of ten data items will be in that node when an observation of the observables is made.

The three fractions above are used to estimate what data items are in the node when the load is estimated. Each data item is multiplied by the load it generates on the node.

Equation 2 gives the resulting formula.

$$\text{estimated load} = \sum_{i=0}^{N} \text{F}(\text{data}_i \text{ hits function}) \times \text{F}(\text{data}_i \text{ hits node with function}) \times \text{F}(\text{data}_i \text{ in node when sampling}) \times \text{L}(\text{data}_i) \quad (2)$$

where N is the number of data items in the topology and $L(\text{data}_i)$ is the load generated by data item i and is measured during the learning phase.

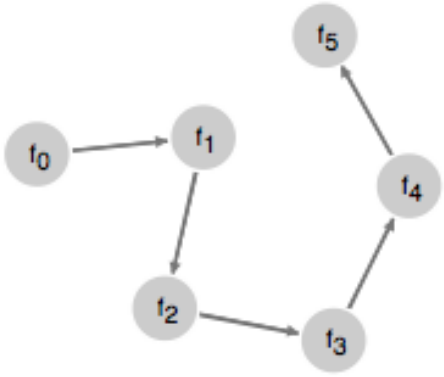


Figure 9: *shows a chain topology.*

3.1.2 Link load estimation

The link load can be estimated as follows:

$$\text{estimated load} = \sum_{i=0}^{N} \text{F}(\text{data}_i \text{ hits link between functions}) \times \text{F}(\text{data}_i \text{ hits this link}) \times \text{F}(\text{data}_i \text{ on link when sampling}) \times \text{L}(\text{data}_i) \quad (3)$$

The link load estimation is similar to the estimation of the node activity parameter. The first fraction returns the fraction of data items of a type traversing a link between two functions. The second fraction returns the fraction of data items of a type traversing the link for which the load is being estimated. As with the node activity estimation, this fraction is dependent on the load balancing behaviour. The third fraction takes into account the time aspect of the topology.

3.1.3 Topology calculation

To detect abnormal behaviour of the topology observable, the maximum number of nodes each function can have, is calculated. Considering that the data items which reside in the network are known, a calculation can be made to find the highest possible amount of nodes per function to process the data items. If the observed amount of nodes of a certain function is greater than this upper bound, then something is amiss in the nodes. This calculation is different from the node activity observable estimation because it

combines the scaling algorithm with the data from the learning phase to detect anomalies.

The calculation of the upper bound is dependent on the implementation of the scaling algorithm because a calculation is made of the worst possible scaling up. If the scaling algorithm scales based upon the memory and CPU load of a node, then this calculation must include those two parameters.

Assuming the scaling only considers the CPU load and the algorithm used is the threshold rules algorithm, the pseudocode in Listing 1 calculates the upper bound.

Listing 1: *This pseudocode calculates the maximum number of nodes a function can have.*

```
1  Input:
2   The function for which the upper bound is calculated
3   The minimum threshold of the scaling algorithm
4   A workflow topology which contains data items
5  Output:
6   The upper bound of the number of nodes a function can have
7  Algorithm (function, minthresh, topology):
8   totalload <-- 0
9   minload <-- minimumLoadPassingThroughFunction(function)
10  if minload < minthresh:
11   minload <-- minthresh
12  endif
13  for all dataitems in topology:
14   if dataitem hits function:
15    totalload <-- totalload + loadOnFunctionByDataItem(function, dataitem)
16   endif
17  endfor
18  numberofnodes <-- totalload / minload
19  if numberofnodes < 1:
20   numberofnodes <-- 1
21  endif
22  return numberofnodes
```

The *minimumLoadPassingThroughFunction* subroutine returns the minimal load a data item generates on that function. For example, a function f_i can process two types of data items, respectively generating a load of 7% and 10%. Then this subroutine would return that the minimal load a data item gives, is 7%. The *loadOnFunctionByDataItem* subroutine returns the load that the data item generates on that function.

The pseudocode of Listing 1 calculates this upper bound by dividing the total load on a function by the minimal load that a function can have, as can be seen on line 18 of the code. The numerator, the total load, is a simplification of the node activity formula, given in Equation 2. The aspect of time and the fraction of the data items passing through a node is discarded. If there is a non-negligible chance of the data item hitting the function then the assumption is made that the data item hits the node at the time the control loop retrieves the actual observables. This will give an upper bound for the total load for a certain function. In other words, it calculates the load on the function if all data items which can hit the function are currently in the function. The denominator uses the minimal threshold as the minimal load to keep a node alive unless all data items which go through the function have a higher load than the minimum threshold. For example, a JPG data item increases the load on the CPU by 10% for the JPG parser, which is higher than the minimal threshold of 5%. Since

this is the only type of data going through this function, every node that is alive must have at least one JPG processing to stay above this threshold. Thus the denominator is 10% instead of the minimal threshold of 5%, giving a lower but still correct upper bound.

Apart from the upper bound calculation, one can look at the relations between the functions. The logical topology of a workflow topology gives these relations. A comparison can be made between the current topology and the logical topology to see if these relations still hold.

3.2 Proof of Concept

To test the estimations of Section 3.1, a proof of concept was developed. The proof of concept was written in Python and implements the workflow topology network, control loop and security capabilities as explained before.

Originally *mininet* was chosen to provide the underlying network for the distributed applications that form the workflow topology. However, after the implementation of the network using *mininet* and primary testing, the proof of concept often became unresponsive. Furthermore, *mininet* does not allow the dynamic creation of nodes during runtime [8]. Alas, the creation of a pool of nodes that can be used and reused was needed. For these two reasons, a full simulator, which simulates the nodes and the links between them, was developed. Topologies and their learning phase are read from configuration files after which the topology is generated by the simulator and data can be send through. This simulator was further used to test the estimation formulas of Section 3.1. One of the advantages of the simulator is that logical time units can be used which can be artificially speed up; allowing to test the same topologies and estimations in a shorter time than in real time.

The first proof of concept was based upon the implementation with *mininet* and implements a multi-threaded environment to simulate communication between nodes. The processing time of data items was implemented by suspending the thread. However, to give the simulator deterministic behaviour, a second proof of concept was implemented without threading but with logical clock ticks to simulate time. This second proof of concept was used to test the estimation formulas and generate the graphs in this thesis.

To allow users to perform actions on the control loop and topology, an API was exposed using Spyne[6]. The proof of concept used this API to create a web interface, as can be seen in Figure 10. This web interface allows to control the amount of data being send through the workflow topology as well as creating a malicious node, leading to a discrepancy in the topology. Furthermore, the control loop can be tweaked. The interval at which the control loop runs, can be modified. The settings of the scaling algorithm can also be tweaked.

[6] `http://spyne.io/`

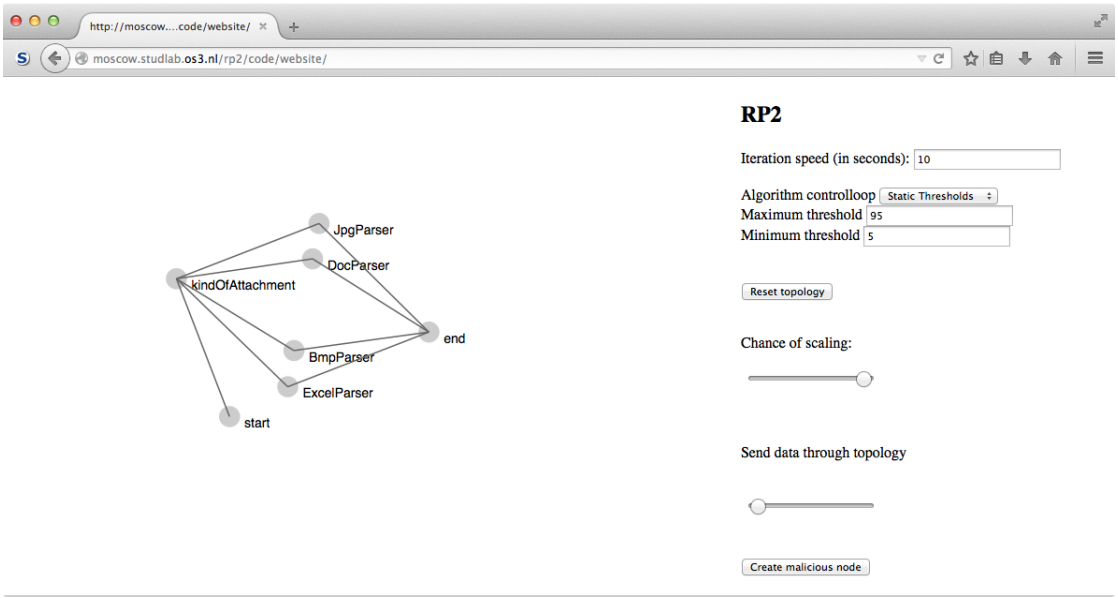


Figure 10: *The web interface of the Proof of Concept. For the visual representation of the topology, D3.js[7]was used. The form on the right allows the user to manipulate the control loop and topology.*

As the node activity parameter, the CPU load was chosen. For the scaling, the static threshold rules algorithm, as defined by Lorido [3], was implemented. A default minimum threshold of 5% and a maximum threshold of 95% for scaling decisions was chosen. These threshold values correspond to the CPU load on the node itself. A CPU load greater than the maximum threshold will result in a scale up of that node. Similarly, a scale down will be performed when the node is lower than the minimum threshold, as defined in Section 2.3.

More information about the code that was developed during this project can be found in Appendix A.

Figure 11: *In the threshold rules algorithm, a scale down is performed when the (CPU) load on the node is lower than the minimum threshold and a scale up is performed when the (CPU) load on the node is higher than the maximum threshold.*

[7] `http://d3js.org/`

4 Results

This section discusses the improvements made to the estimation formulas and results of how correct the estimations are.

4.1 Improvements to estimations

In early test runs of the proof of concept the estimation formulas, as explained in Section 3.1, gave several false positives related to previous data streams on the topology. One example of such a problem is calculating the upper bound of the topology. When there was previously a significantly higher load, the control loop will have scaled up the topology. Consider the case of a high load on the network and a scale out being performed by the control loop. If subsequently the load drops and no more data items are processed; when an estimation of the maximum number of nodes is made, it takes into account the data items that reside in the network, which is none and thus an underestimation is made of the maximum number of nodes in the network. Thus, we must consider the timings of the scaling algorithm such that the topology has scaled down from previous loads before estimations of the upper bound are made.

To decrease the number of false positives for all observables, a shift register was implemented which contains previous discrepancies. This shift register contains entries for the last runs of the control loop. Whether or not a discrepancy is found between the estimated value and the observed value, is stored in the shift register. Only when all entries in the register show a discrepancy between the estimated load and the real load, is the autonomous response by the control loop triggered. When the shift register has been filled, a new run of the control loop will delete the oldest entry in the register and insert the new value.

The size of this register for the node activity and link load observable is dependent on the total time a data item spends in the topology and the speed of the control loop. This is necessary so that the data items can traverse the whole topology before an autonomous response is triggered. If we consider a chain of functions, as in Figure 9, and a burst of data going through this chain then the data will hit the last function after a certain amount of time. If the control loop executes when the data has not yet reached this function then the estimation will overestimate the real load, which is zero, leading to an alert. Multiple measurements will filter this false positive out. This register is implemented for every node and link, and the size can differ between different nodes because the total time inside the topology might differ greatly. However, the proof of concept uses one size, namely the division between the longest time a data item can be processed divided by the interval of the control loop. In the case of the proof of concept, the maximum time a data item is processed is 7 time units and the control loop interval is 5 time units, giving the register a size of two.

To at least compensate for minor fluctuations in the observables in respect to the learning phase, the minimal size of the array in the proof of concept was chosen to be three. This is only of importance in small topologies where the total time needed for data items to traverse the topology is marginally bigger (or even smaller) than the control

loop interval, resulting in an array of size one or two. Using this minimal size, these small topologies can have the advantage of not triggering on sudden discrepancies.

4.2 Results of estimations on example topology

The topology used to check if the estimation formulas can be used to estimate the real loads was already mentioned in Section 2.2 and consists of functions to parse email attachments. It is a simple example to show the workings of the estimation formulas and has the additional advantage of not being as abstract as a topology with functions f_0 to f_n.

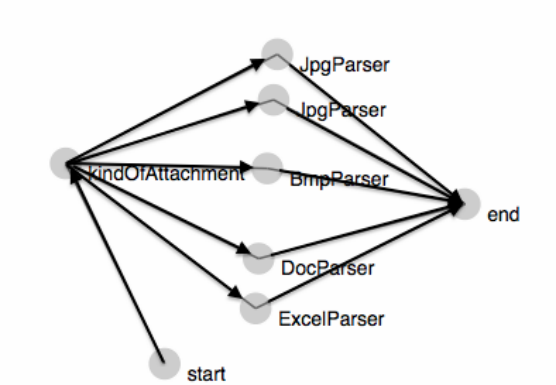


Figure 12: *The topology consists of a router function (kindOfAttachment), several parser functions (JpgParser, DocParser, ExcelParser and BmpParser) and two controller nodes (start and end).*

Every type of data corresponds to a parser in the topology and has 1:1 fraction of going through that parser. The *kindOfAttachment* function takes 2 time units to process a data item and every data item gives a load of 2% of the total CPU capacity on said function. The discrepancy limit has been set to 20%, meaning that only if the estimated load differs by more than 20%, a discrepancy is noted in the register of that observable.

Table 1 shows the learned metadata about the topology. The topology handles four data types: JPG, DOC, XLS and BMP. Each data item has a corresponding predicted CPU load on the corresponding parser function and a certain amount of time needed to process the data item.

Data type	**Predicted CPU load**	**Time to process**
JPG	10 %	5 time units
DOC	10 %	5 time units
XLS	15 %	5 time units
BMP	10 %	5 time units

Table 1: *For each data type there is a predicted CPU load and time needed to process the data item on the corresponding parser.*

Through this topology, a continuous load of eight JPGs, three documents (DOC) and two excel (XLS) documents are distributed at each logical time tick. This distribution of data items follows a push technique. Every function pushes data to the next function. The interval between subsequent runs of the control loop, including the auto-scaling, was set at 5 time units.

Figure 13 shows the estimation of the node activity in comparison with the actual loads of the nodes. Several scale ups are performed by the topology to handle the load, these coincide with the runs of the control loop, each 5 time units. To get a better view of the estimations made by the control loop, the estimations were executed every clock tick. The nodes have been given a load of zero in the graph for the time units in which they do not exist.

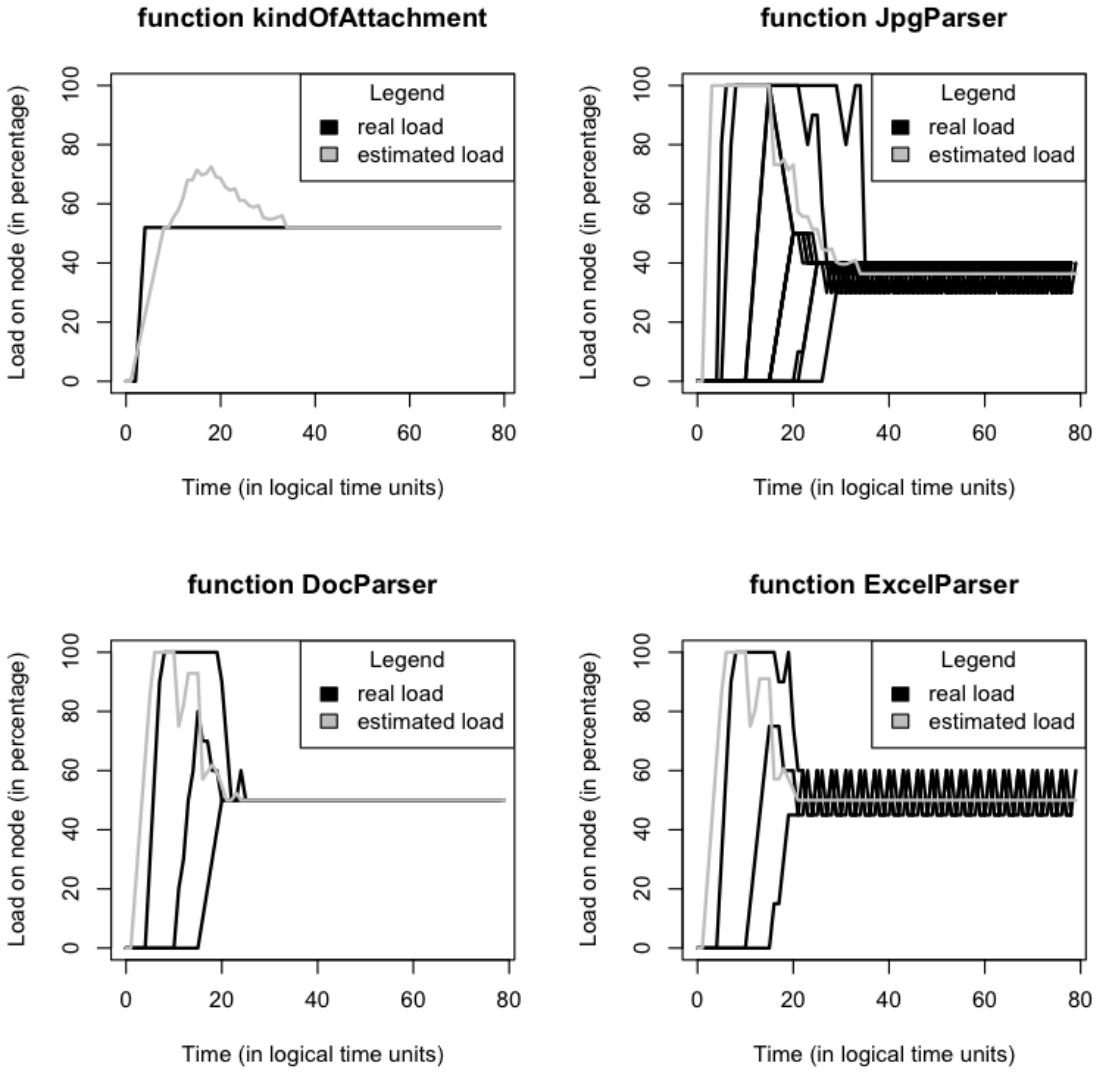


Figure 13: *For each function, the estimation and real load are plotted. Each line represents a node with that function. The graphs show the control loop scaling up the function every five seconds until congestion is resolved.*

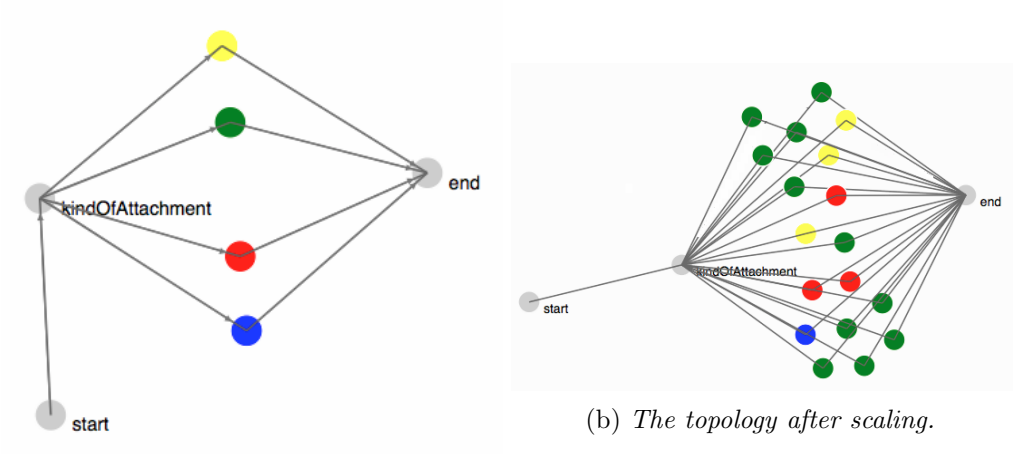
The left top graph of Figure 13 shows the load on the nodes with the *kindOfAttachment* function. The overestimation of the estimated load between $t = 10$ and $t = 33$ is the result of congestion in other functions. This congestion in other functions results in more data items in the topology and more specifically in those other functions. As such, the time fraction of Equation 2 is an overestimation of the free flow time fraction. The total time a data item spends in the topology, the denominator of the time fraction, is longer than estimated due to congestion. After the congestion is resolved in the other functions and a free-flow network is established, the estimation accuractely predicts the load in the node.

The right top graph shows the JPG parser function that has the most processing to do. Only after five iterations of the control loop, resulting in ten additional nodes, does the topology stabilize. After the topology stabilizes, the estimated load prediction falls in between the actual loads seen on the nodes of the JPG parser. This difference in real loads between nodes is caused by the undividability of the data items. One data item generates a certain load on a function, due to the round-robin load balancing, some nodes get one data item more than other JPG parsers. This is exactly what happens in the graph as there are eleven JPG parser nodes but at each time tick only eight JPG data items are distributed between those nodes.

The left bottom graph in figure 13 shows another accurate prediction of the load once the function was scaled up to deal with congestion. Once the congestion is resolved, there are 21 document data items in the topology which pass through the DOC parser. Since the topology scaled up to three nodes, every node gets a fraction of one in three DOC data items, each one generating a load of 10% of the total CPU load. Because of the time aspect, only five out of seven data items will generally be in the parser as a data item is processed for 5 time ticks in the function and the total time the data item spends in the network is 7 time ticks. Multiplying all these elements ($1 * \frac{1}{3} * \frac{5}{7} * 10 * 21$) gives us an estimated load of 50% on a document parser node. All other data items in the workflow topology do not contribute to the estimated load of the nodes with this function because they do not flow through this function.

The right bottom graph shows similar behaviour as the JPG parser function as two data items are distributed over three nodes in a round-robin fashion. The observed loads, after stabilisation, are either 45% or 60% and an oscillation between these two is seen because of the round-robin load balancing. This difference of 15% is exactly the load which one data item generates on the node.

All graphs of Figure 13 show that once the congestion in the workflow topology is resolved, by scaling up the topology and arriving at a stable topology, the estimation becomes more accurately, even to the point of being the same as the real load in the case of the DOC parser and *kindOfAttachment* function. Of course, as this is done in a simulator and not on a real network, the estimations are made in the best possible environment. Real networks might have bigger differences between the estimated load and the observed load.



(a) *The logical topology.*

(b) *The topology after scaling.*

Figure 14: *The logical topology and the topology after convergence to handle the load generated by the data. Green nodes signify JPG parsers, yellow nodes DOC parsers, red nodes Excel parsers and blue nodes BMP parsers.*

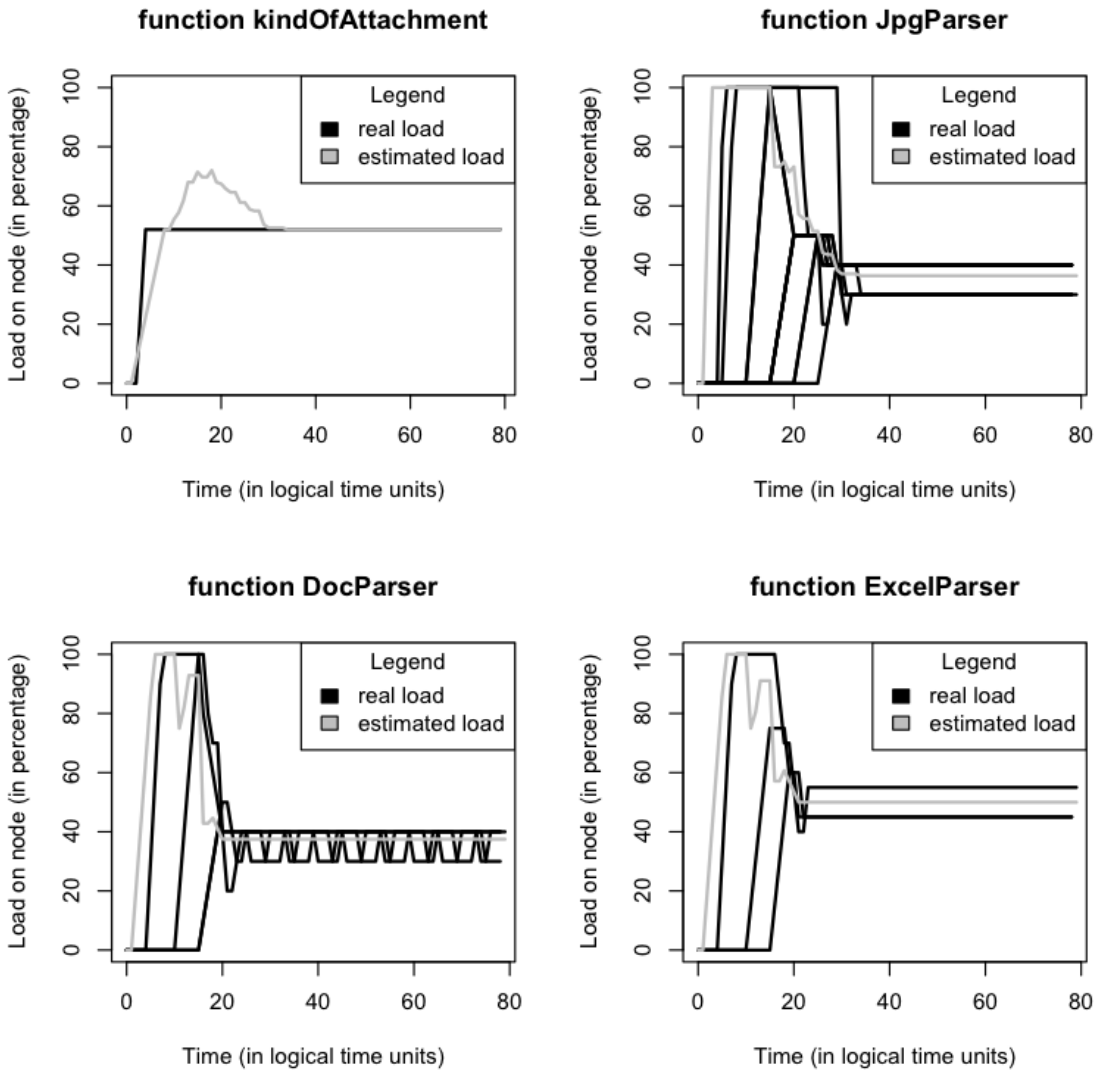


Figure 15: *A load ordered round-robin distribution in the topology gives different loads and estimations on the nodes than a normal round-robin load balancing technique.*

Figure 14b shows the topology generated by the control loop and the data. The three parsers that received data items have all been scaled up. The JPG parser scaled up to eleven nodes, the DOC parser to three nodes and the Excel parser to three nodes.

Figure 15 shows the influence of using a different load balancing technique to push data items, namely a load ordered round-robin. A load ordered round-robin performs an ordering of its neighbours according to the current load they have and only then sends the data items in a round-robin distribution. As such, when there are more nodes than data items to distribute, this ordering will not give any data items to the most overloaded nodes. This results in a lower load on these nodes because no new data items were distributed to them. Compared to regular round-robin, this ordering allows for faster convergence. This can be seen when comparing the JPG parser graphs of the two load balancing techniques.

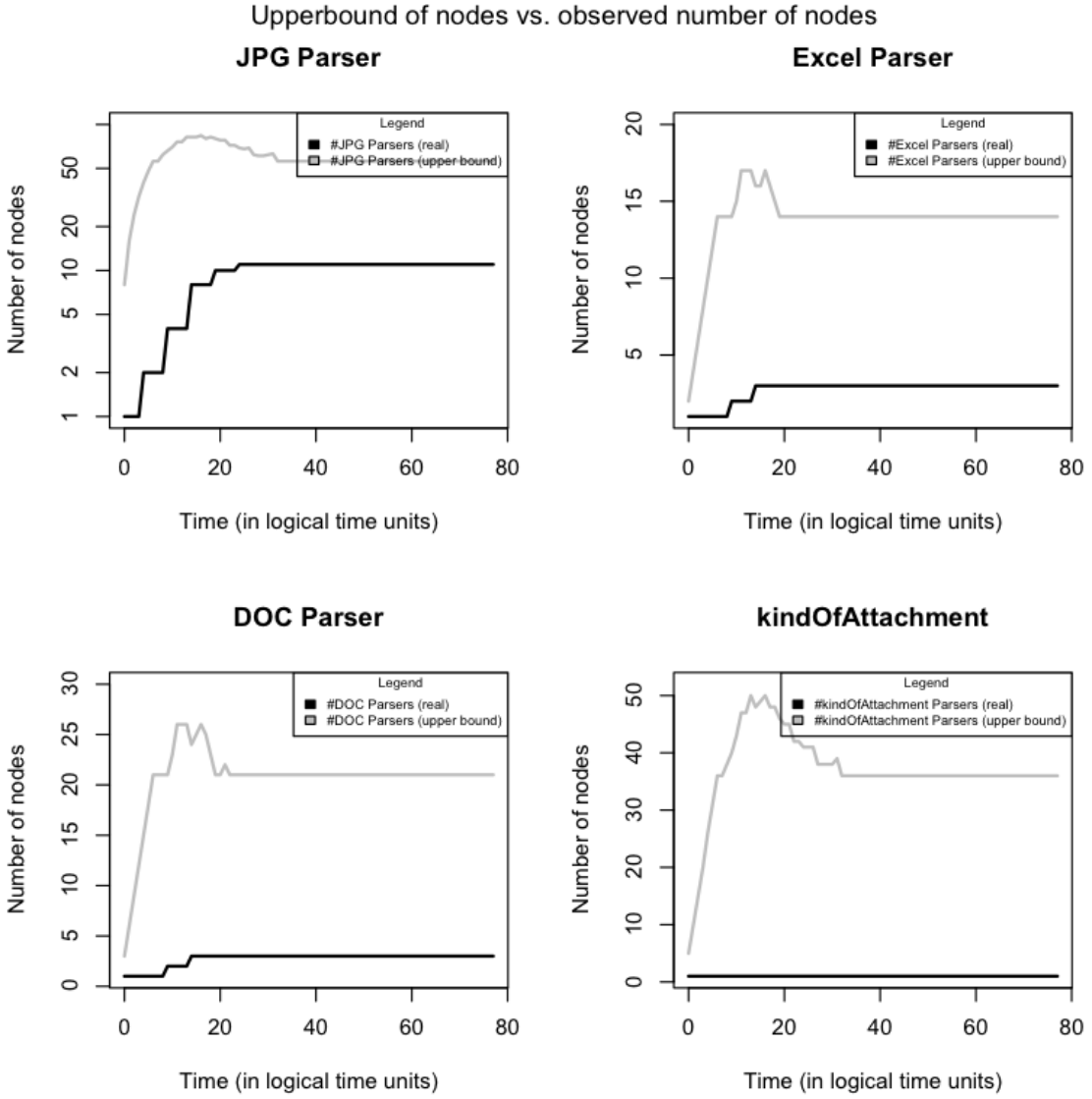


Figure 16: *shows the maximum number of nodes per function, calculated by knowing the data item which reside in the topology, and the observed number of nodes per function.*

Figure 16 shows the number of nodes that a function has in relation to the time. The upper bound is also plotted as a function of time. The graph shows that all upper bounds are indeed upper bounds for their observed counterpart in the workflow topology. However, it also shows that there is a significant difference between the observed value and the upper bound, leading one to think about its usefulness. The *kindOfAttachment* upper bound is at least 36 times the actual value once the first data items have finished processing (after $t = 7$). This overestimation is the result of all data types having a non-zero fraction of passing through this function and the low load the data items generate on the function. Once the congestion is resolved, there are 91 data items in the topology. Given that the generated load on the processor per data item is 2% of the total CPU capacity on the *kindOfAttachment* function, this would mean that in the worst case, when all data items reside in the function, there would be a total load of 182%, enough to keep 36 ($\frac{182}{5}$, where the denominator is the minimum threshold) nodes alive without the scaling algorithm pruning nodes.

There are no results for the link load observable because the implementation and simulation of this observable was not finished during this project. As the link load estimation is similar to the node activity estimation, the expectation is that it will show similar results.

4.2.1 Creating a security issue

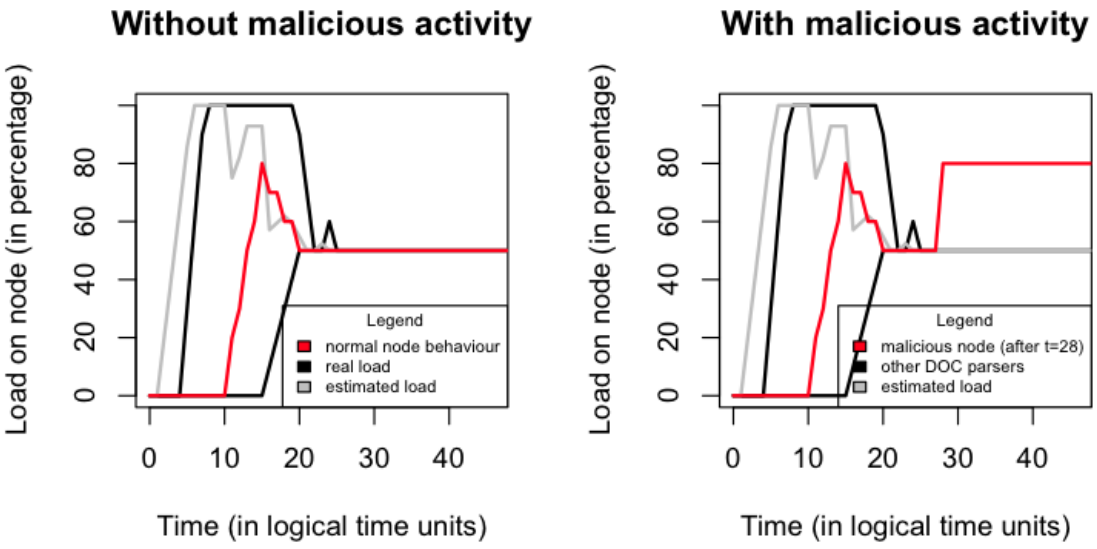
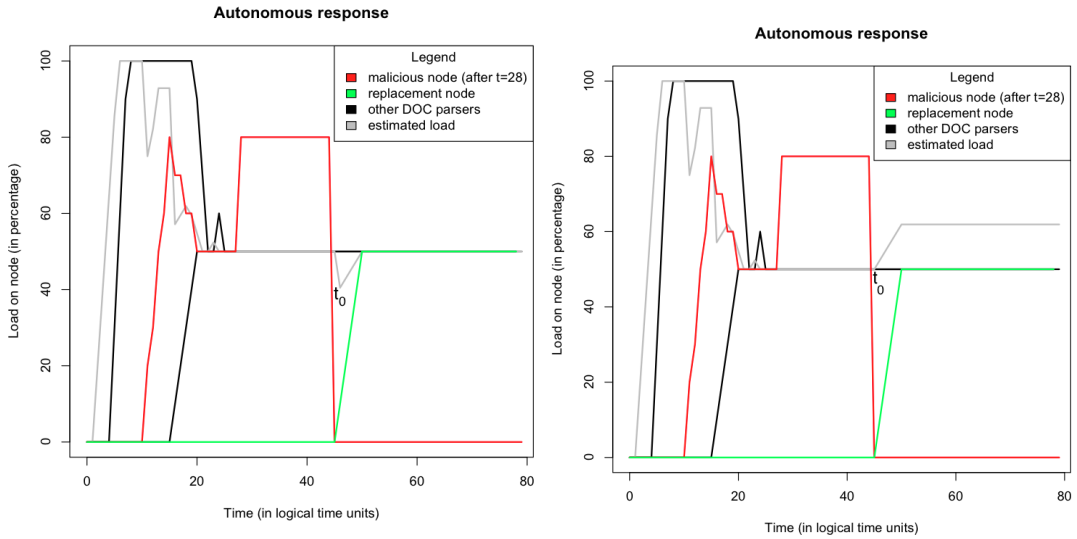


Figure 17: *A sudden increase of the CPU load of a node can be seen in the right graph. As the difference between the observed load and estimated load is bigger than the discrepancy limit (20%), an alert in the control loop will be triggered.*

As an example security violation, the node activity of one of the nodes was increased to report a CPU load of 80%[8]. Figure 17 shows the sudden increase of load at time $t = 28$.

[8] `https://www.youtube.com/watch?v=X5ohVjI7OuQ` contains a visual representation of the control loop intercepting the malicious node.

Figure 18 shows the response of the network in action. Because the discrepancy is larger than the limit set by the network, 20%, the shift register will be filled with alerts. At $t = 45$, all entries in the register show an alert and the network responds by creating a new node and killing the malicious node. This can be seen in the graph by the decreasing node at $t = 45$ and the increasing node at $t = 45$.



(a) *When a node is killed, the node sends the data items to the end of the topology (unfinished).*

(b) *When a node is killed, the data items stay in the topology, according to the control loop.*

Figure 18: *The malicious node (shown in red) gets killed by the control loop. At the same time, a new node (shown in green) is spawned, which can take over the work of the malicious node. The estimated load after the killing of the node depends on the implementation of how data items that are being processed are dealt with.*

Figure 18 shows a node, which has abnormal behaviour, being killed by the control loop. After killing the node, a new node with the same function is spawned to take the place of the node with abnormal behaviour.

Figure 18a shows a decrease of data items in the topology at t_0, shown by a drop in the estimated load. This is the effect when the data items are cleaned up by the node before destruction, thereby temporarely decreasing the amount of data in the topology. In this case, the assumption is made that the clean up consists of sending unfinished data items to the *end* function.

Figure 18b shows an increase in the estimated load, at t_0, after the malicious node was killed. This behaviour is the result of the data items, which were processed on the node, getting killed together with the node. Although these data items are no longer in the topology, the control loop considers them for the estimation as it has not yet seen these data items at the end of the topology and thus makes the assumption that the data items are still being processed, leading to an overestimation of the load.

Depending on how the malicious node is killed in the workflow topology, one of these situations applies. In the case of the lost data items, a timeout can be implemented by the control loop to discard data items that have been in the workflow topology for too long, thereby making the estimation load more accurate again.

5 Discussion and future work

This section covers several points that were not yet fully discussed in this thesis and can be considered as future work.

A first discussion point is the importance of the learning phase, discussed in Section 3. If the values in the learned phase do not apply to the topology, then the estimations made, will be off and thus trigger more false positives. Because of the discrepancy limit, the amount of false positives can be kept low for small differences between what is learned and what is measured. Section 3 also mentions the relation between the amount of data and the capabilities of the control loop. Thus instead of having a fixed value (e.g. average) for the load on a node or link, as was the assumption in the proof of concept, one can also learn the variance around that value, allowing the control loop to better predict anomalous behaviour. How the estimations can be improved further is an interesting point of investigation.

A second discussion point is the trust put into the actual values given by topology. When a node is compromised, do we still trust the parameter values given? Do we trust the actual CPU load, in the case of the proof of concept? What parts of the node do we trust (operating system, firmware) and what not? A decision should be made on basis of the types of attacks that are expected and what the likelihood of such an attack is. As this is application dependent, this thesis has not looked into this discussion.

A third discussion point is the speed of sampling. This thesis has made the assumption, as in Equation 1, that a difference can be taken between the start and the end of the topology to get the data in between. In real life topologies, a time difference in the sampling of the start and end node can occur, resulting in an inaccurate overview of the data in the topology. Depending on the variability in the types of data and the differences between those types, this can be an issue for the estimations. How big this time difference can be and what the causes are, is another point for future investigation.

A final discussion point is the congestion of the nodes; the results show that the estimations are more accurate when there is free flow in the network. It is thus important that congestion is resolved as quickly as possible. This could be done by redistributing queued data items after a scale up is performed by the control loop. Another possibility is a more aggressive scale up to resolve the congestion faster. Other solutions and which one performs better is considered future work.

6 Conclusion

This project focused on the use of workflow topology observables in security autonomous response networks. To answer how these could be used, four sub questions were defined.

The first sub question asked what the characteristic observables are. Observables were defined as properties of the topology that can be monitored by the control loop. The topology itself, the loads on the link and the activity on the nodes as a result of data flowing through the topology were discussed as observables of workflow topologies.

The second sub question looked at the sensitivity of observables. In other words, what causes the observables to change? The proof of concept showed that the data sent through the network changes all three observables discussed. The load balancing technique is also important as it distributes the data items over nodes with the same function, thereby influencing the workflow topology observables. The proof of concept showed that the observables change when a different load balancing technique is used. Another element that influences the observables is the scaling algorithm as it decides when to create and delete nodes.

The third sub question posed the question of what kind of attacks can be detected by looking at the observables and how one can respond. The control loop can detect discrepancies in the load generated on links and on nodes, detecting unusual usage of resources. Furthermore, the control loop checks the relations between functions such that excessive communication between nodes that do not have a relation is detected. We looked into the countermeasures which can be taken and discussed the use of respawning, sandboxing, honeypotting and reprovisioning for the node activity observable and the use of flow entries for the link load observable.

Using this research, we looked at how we could use the observables in a security context. We used the data residing in the topology, which influences the observable, to estimate the observables and compare it to the actual observables observed in the workflow topology. To estimate the observables, we used data learned from a learning phase. The proof of concept implemented a simulator which showed that the estimations predict the actual observables. The next step, however, is to verify the estimations and their accuracy on real networks.

References

[1] Greg Ferro, "Response: Customer Intent on SDN Adoption is Accelerating with 85% Adoption by 2016." `http://etherealmind.com/response-customer-intent-sdn-adoption-accelerating-85-adoption-2016/`, 2014.

[2] M. X. Makkes, R. Cushing, A. Belloum, S. Olabarriaga, M. Baranowski, C. de Laat, and R. Meijer, "Data Intrinsic networked computing," 2015.

[3] Tania Lorido-Botrán, José Miguel-Alonso, José A. Lozano, "Auto-scaling Techniques for Elastic Applications in Cloud Environments." http://www8.cs.umu.se/kurser/5DV153/HT14/literature/lorido2012autoscaling.pdf, 2012.

[4] Karamjeet Kaur, Japinder Singh and Navtej Singh Ghumman, "Mininet as Software Defined Networking Testing Platform ." http://www.sbsstc.ac.in/icccs2014/Papers/Paper29.pdf, 2014.

[5] Hristo Dimitrov, "Implementing Security Control Loops in Security Autonomous Response Networks." http://rp.delaat.net/2013-2014/p13/report.pdf, 2014.

[6] United States Computer Emergency Readiness Team (US-CERT), "Alert (TA14-017A) UDP-based Amplification Attacks." https://www.us-cert.gov/ncas/alerts/TA14-017A, 2014.

[7] Chiosi et al., "Network Functions Virtualisation." https://portal.etsi.org/NFV/NFV_White_Paper.pdf, 2012.

[8] Jason Parraga, "Creating hosts dynamically in mininet." https://groups.google.com/a/openflowhub.org/forum/#\protect\kern-.1667em\relaxtopic/floodlight-dev/_RhkNz98Nc4, 2013.

7 Appendix A: Codebase

For the proof of concept, code was written to get a visual representation of the workflow topologies and to test the estimations. Because of the size of the codebase, the codebase, including the version control, was given to the System and Network Engineering Master at the University of Amsterdam.

To create the graphs shown in this paper, the following commits were used:

Figure	Branch	Commit ID	CLI	Topology
Figure 13	goingsequential	6d10d5fff5	`python main.py overview`	`code/topology5.yaml`
Figure 15	goingsequential	d78713614b	`python main.py overview`	`code/topology5.yaml`
Figure 16	goingsequential	6d10d5fff5	`python main.py overview`	`code/topology5.yaml`
Figure 17a	goingsequential	6d10d5fff5	`python main.py overview`	`code/topology5.yaml`
Figure 17b	goingsequential	6d10d5fff5	`python main.py malicious`	`code/topology5.yaml`
Figure 18a	goingsequential	6d10d5fff5	`python main.py malicious`	`code/topology5.yaml`
Figure 18b	goingsequential	2a5e942166	`python main.py malicious`	`code/topology5.yaml`