Usage of Iterated Local Search to Improve Firewall Evolvability

Haerens Geert
Antwerp University, Engie
Brussels, Belgium
email: geert.haerens@engie.com

***Abstract*—The Transmission Control Protocol/Internet Protocol (TCP/IP) based firewall is a notorious non-evolvable system. Changes to the firewall often result in unforeseen side effects, resulting in the unavailability of network resources. The root cause of these issues lies in the order sensitivity of the rule base and hidden relationships between rules. It is not only essential to define the correct rule. The rule must be placed at the right location in the rule base. As the rule base becomes more extensive, the problem increases. According to Normalized Systems, this is a Combinatorial Effect. In previous research, an artifact has been proposed to build a rule base from scratch in such a way that the rules will be disjoint from each other. Having disjoint rules is the necessary condition to eliminate the order sensitivity and thus the evolvability issues. In this paper, an algorithm, based on the Iterated Local Search metaheuristic, will be presented that will disentangle the service component in an existing rule base into disjoint service definitions. Such disentanglement is a necessary condition to transform a non-disjoint rule base into a disjoint rule base.**

Keywords–Firewall; Rule Base; Evolvability; Metaheuristic; Iterated Local Search.

I. INTRODUCTION

The TCP/IP based firewall has been and will continue to be an essential network security component in protecting network-connected resources from unwanted traffic. The increasing size of corporate networks and connectivity needs has resulted in firewall rule bases increasing considerably. Large rule bases have a nasty side effect. It becomes increasingly difficult to add the right rule at the correct location in the firewall. Anomalies start appearing in the rule base, resulting in the erosion of the firewall's security policy or incorrect functioning. Making changes to the firewall rule base becomes more complex as the size of the system grows. An observation shared by Forrester [1] and the firewall security industry [2] [3]. A more detailed literature review on the topic can be found in [4].

Normalized Systems (NS) theory [5] defines a Combinatorial Effect (CE) as the effect that occurs when the impact of a change is proportional to the nature of the change and the system's size. According to NS, a system that suffers from CE is considered unstable under change or non evolvable. A firewall suffers from CE. The evolvability issues are the root cause of the growing complexity of the firewall as time goes by.

The order sensitivity plays a vital role in the evolvability issues of the rule base. The necessary condition to remove the order sensitivity is known, being non-overlapping or disjoint rules. However, firewall rule bases don't enforce that condition, leaving the door open for misconfiguration. While previous work investigates the causes of anomalies [6] [7], detecting anomalies [8] [9] [10] and correcting anomalies at the time of entering new rules in the rule base [8], to the best of our knowledge and efforts, no work was found that tries to construct a rule base with ex-ante proven evolvability (= free of CE). While previous methods are reactive, this paper proposes a proactive approach.

Issues with evolvability of the firewall rule base induce business risks. The first is the risk of technical communication paths not being available to execute business activities properly. The second is that flaws in the rule base may result in security issues, making the business vulnerable for malicious hacks resulting in business activities' impediment.

In this paper, we propose an artifact, an algorithm, that aims at converting an existing non-evolvable rule base into an evolvable rule base. Design Science [11] [12] is suited for research that wants to improve things through artifacts (tools, methods, algorithms, etc.). The Design Science Framework (see Figure 1) defines a relevance cycle (solve a real and relevant problem) and rigor cycle (grounded approach, usage of existing knowledge and methodologies).

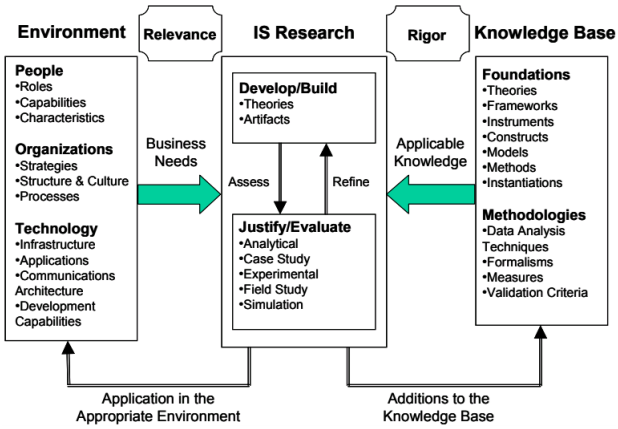

Figure 1. The Design Science Framework - from [11] .

The Design Science Process (see Figure 2) guides the artifact creation process according to the relevance and rigor cycle. What follows is structured according to the Design Science process.

Section II introduces the basic concepts of firewalls, fire-

 ISBN: 978-1-61208-850-1

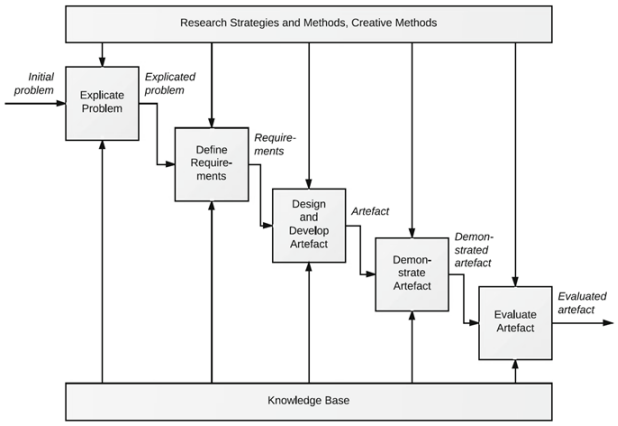

Figure 2. The Design Science Process - from [12].

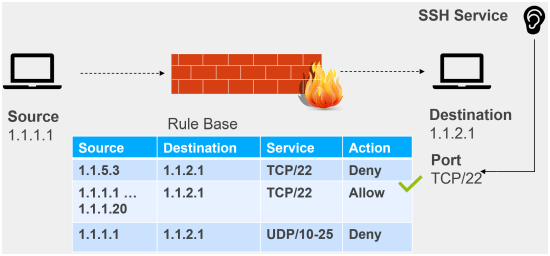

Figure 3. Firewall concepts.

wall rule relationships, Normalized Systems, and the evolvability issues of the firewall. In Section III, we will discuss the requirements for an algorithm that will transform a non-evolvable rule base, into an evolvable rule base. Section IV will build the different components of the proposed algorithm using the Iterated Local Search metaheuristic. In Section V, the algorithm will be demonstrated in a number of cases. In Section VI, we evaluate and discuss our findings and we wrap-up with a conclusion in Section VII.

II. Problem Description

The first part of this section will explain how a firewall works and the concept of firewall group objects. The second part will discuss the relationships between firewall rules and introduces the Normalized Systems theory.

A. Firewall basics

An Internet Protocol Version 4 (IP4) TCP/IP based firewall, located in the network path between resources, can filter traffic between the resources, based on the Layer 3 (IP address) and Layer 4 (TCP/UDP ports) properties of those resources [13]. UDP stands for User Datagram Protocol and is, next to TCP, a post based communcation protocol at the 4th level of the Open Systems Interconnection Model (OSI Model) [14]. Filtering happens by making use of rules. A rule is a tuple containing the following elements: <Source IP, Destination IP, Protocol, Destination Port, Action>. IP stands for IP address and is a 32-bit number that uniquely identifies a networked resource on a TCP/IP based network. The protocol can be TCP or UDP. Port is a 16-bit number (0 - 65.535) representing the TCP or UDP port on which a service is listening on the 4th layer of the OSI-stack. When a firewall sees traffic coming from a resource with IP address =<Source IP>, going to resource =<Destination IP>, addressing a service listening on Port = <Destination port>, using Protocol = <Protocol>, the firewall will look for the first rule in the rule base that matches Source IP, Destination IP, Protocol and Destination Port, and will perform an action = <Action>, as described in the matched rule. The action can be “Allow” or “Deny”. See Figure 3 for a graphical representation of the explained concepts.

A firewall rule base is a collection of order-sensitive rules. The firewall starts at the top of the rule base until it encounters the first rule that matches the traffic. In a firewall rule, <Source IP>, <Destination IP>, <Destination Port> and <Protocol> can be one value or a range of values. In the remainder of this paper, protocol and port are grouped together in service (for example, TCP port 58 or UDP port 58 are 2 different services).

B. Firewall group objects

Rules containing IP addresses for source/destination and port numbers, are difficult to interpret by humans. Modern firewalls allow the usage of firewall objects, called groups, to give a logical name to a source, a destination, or a port, which is more human-friendly. Groups are populated with IP addresses or ports and can be nested. The groups are used in the definition of the rules. Using groups should improve the manageability of the firewall.

C. Firewall rule relationships

In [6], the following relations are defined between rules:

- **Exactly Matching:** Exactly matching rules (**Rx=Ry**). Rules **Rx** and **Ry** are exactly matched if every field in **Rx** is equal to the corresponding field in **Ry**.
- **Inclusively Matching:** Inclusively matching rules (**Rx⊂Ry**). Rule **Rx** inclusively matches **Ry** if the rules do not exactly match and if every field in **Rx** is a subset or equal to the corresponding field in **Ry**. **Rx** is called the subset match while **Ry** is called the superset match.
- **Correlated**: Correlated rules (**Rx⋈Ry**). Rules **Rx** and **Ry** are correlated if at least one field in **Rx** is a subset or partially intersects with the corresponding field in **Ry**, and at least one field in **Ry** is a superset or partially intersects with the corresponding field in **Rx**, and the rest of the fields are equal. This means that there is an intersection between the address space of the correlated rules although neither one is subset of the other.
- **Disjoint:** Rules **Rx** and **Ry** are completely disjoint if every field in **Rx** is not a subset and not a superset and not equal to the corresponding field in **Ry**. However, rules **Rx** and **Ry** are partially disjoint if there is at least one field in **Rx** that is a subset or a superset or equal to the corresponding field in **Ry**, and there is at least one field in **Rx** that is not a subset and not a superset and not equal to the corresponding field in **Ry**.

 ISBN: 978-1-61208-850-1

Figure 4 represents the differnet relations in a graphical manner. Exactly matching, inclusively matching and correlated rules can result in the following firewall anomalies [8]:

- *Shadowing Anomaly*: A rule **Rx** is shadowed by another rule **Ry** if **Ry** precedes **Rx** in the policy, and **Ry** can match all the packets matched by **Rx**. The result is that **Rx** is never activated.
- *Correlation Anomaly*: Two rules **Rx** and **Ry** can cause a collerlation anomaly if, the rules **Rx** and **Ry** are correlated and if **Rx** and **Ry** have different filtering actions.
- *Redundancy Anomaly*: A redundant rule **Rx** performs the same action on the same packets as another rule **Ry** so that if **Rx** is removed the security policy will not be affected.

A fully consistent rule base should only contain disjoint rules. In that case, the order of the rules in the rule base is of no importance, and the anomalies described above will not occur [6] [7] [8] . However, due to several reasons such as unclear requirements, a faulty change management process, lack of organization, manual interventions, and system complexity [13], the rule base will include correlated, exactly matching, and inclusively matching rules, and thus resulting in evolvability issues.

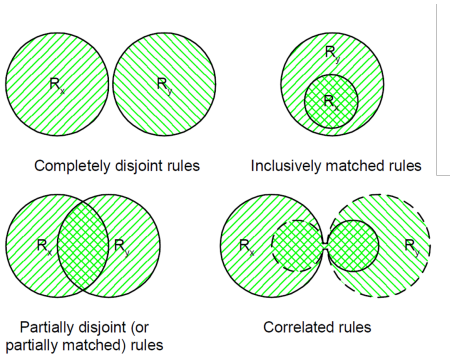

Figure 4. Possible relationships between rules (from [9]).

D. Normalized Systems concepts

Normalized Systems theory [5] [15] originates from the field of software development.

The Normalized Systems Theory takes the concept of system theoretic stability from the domain of classic engineering to determine the necessary conditions a modular structure of a system must adhere to in order for the system to exhibit stability under change. Stability is defined as Bounded Input equals Bounded Output (BIBO). Transferring this concept to software design, one can consider bounded input as a certain amount of functional changes to the software and the bounded output as the number of effective software changes. If the amount of effective software changes is not only proportional to the amount of functional changes but also the size of the existing software system, then NS states that the system exhibits a CE and is considered unstable under change.

Normalized Systems Theory proves that, in order to eliminate CE, the software system must have a certain modular structure, were each module respects four design rules. Those rules are:

- Separation of Concern (SoC): a module should only address one concern or change driver.
- Separation of State (SoS): a state should separate the use of a module by another module during its operation.
- Action Version Transparency (AVT): a module, performing an action should be changeable without impacting modules calling this action.
- Data Version Transparency (DVT): a module performing a certain action on a data structure, should be able to continue doing this action, even is the data structures has undergone change (add/remove attributes).

NS can be used to study evolvability in any system, which can be seen as a modular system and derive design criteria for the evolvability of such a system [16] [17].

III. Requirements for the Solution

This section will discuss the design requirements for an evolution rule base built from the ground up, also known as a green-field approach. These design requirements serve as input for a brown-field approach or convert a non-evolvable rule base into an evolvable rule base.

A. Building an Evolvable Rule Base

In previous work [4], the combinatorics involved when creating a rule base are discussed. For a given network **N**, containing $\mathbf{C_j}$ sources and $\mathbf{H_j}$ destinations, offering 2^{17} services (protocol/port), and having a firewall **F** between the sources and the destinations, it can be shown that $\mathbf{f_{max}}$ is the number of possible rules that can be defined on the firewall **F**:

$$\mathbf{f_{max}} = 2.\left(\sum_{a=1}^{Hj}\binom{Cj}{a}\right).\left(\sum_{a=1}^{Hj}\binom{Hj}{a}\right).\left(\sum_{k=1}^{2^{17}}\binom{2^{17}}{k}\right) \quad (1)$$

where $\mathbf{C_j}$ and $\mathbf{H_j}$ are function of **N**: $\mathbf{C_j} = f_c(\mathbf{N})$ and $\mathbf{H_j} = f_h(\mathbf{N})$

A subset of those rules will respresent the intended security policy and only a subset of that subset will be the set of rules that are disjoint. The maximum size of the disjoint set of "allow" rules (aka a white list) is:

$$\mathbf{f_{disjoint}} = \mathbf{H_j}.2^{17} \quad (2)$$

with $\mathbf{H_j}$ is the number of hosts connected to the network. $\boldsymbol{H_j} = \boldsymbol{fh(N)}$ and 2^{17} the max amount of services available on a host.

The probability that a firewall administrator will always pick rules from the disjoint set is low if there is no conscious design behind the selection of rules.

In previous work [4], based on NS, an artifact is being proposed to create a rule base free of CE for a set of anticipated

changes. The artifact takes the "Zero Trust" [**?**] [**?**] design criteria into account as well, meaning that access is given to the strict minimum: in this case, the combination of host and service.

1) Starting from an empty firewall rule base **F**. Add as first rule the default deny rule **F**[1]= $\mathbf{R}_{\text{default\_deny}}$ with
   - $\mathbf{R}_{\text{default\_deny}}$.Source = ANY,
   - $\mathbf{R}_{\text{default\_deny}}$.Destination=ANY,
   - $\mathbf{R}_{\text{default\_deny}}$.Service= ANY,
   - $\mathbf{R}_{\text{default\_deny}}$.Action = "Deny".
2) For each service offered on the network, create a group. All service groups need to be completely disjoint from each other: the intersection between groups must be empty.
   Naming convention to follow:
   - **S**_*service.name*,
   - with *service.name* as the name of the service.
3) For each host offering the service defined in the previous step, a group must be created containing only one item (being the host offering that specific service).
   Naming convention to follow:
   - **H**_*host.name*_**S**_*service.name*,
   - with *host.name* as the name of the host offering the service
4) For each host offering a service, a client group must be created. That group will contain all clients requiring access to the specific service on the specific host.
   Naming convention to follow:
   - **C_H**_*host.name*_**S**_*service.name*
5) For each **S**_*service.name*,**H**_*host.name*_**S**_*service.name* combination, create a rule **R** with:
   - **R**.Source =**C_H**_*host.name*_**S**_*service.name*
   - **R**.Destination = **H**_*host.name*_**S**_*service.name*
   - **R**.Service= **S**_*service.name*
   - **R**.Action = "Allow"

   Add those rules to the firewall rule base **F**.
   The default rule $\mathbf{R}_{\text{default}}$ should always be at the end of the rule base.

In [4], proof can be found that a rule base created according to the above artifact, results in an evolvable rule base concerning the following set of anticipated changes: addition/removal of a rule, addition/removal of a service, addition/removal of a destination (with or without a new service), the addition of a source. The removal of a source does not impact the rule base but does impact the groups containing the sources.

B. Converting an Existing Rule Base into an Evolvable Rule Base

The previous section describes the green-field situation; builiding a rule base from scratch. The luxery of a green-field is often not present. We require a solution that can convert an existing rule base, into a rule rule base that only contains disjoint rules. Of couse, the original filtering strategy expressed in the rule base must stay the same. From the previous section we know that we require disjoint service definitions. If we can disentangle the service defintions, and adjust the rules accordingly, we have our basic building block for a disjoint rule base. For each disjoint service definition, we need to create as many destination groups as there are host offering that service (lookup in rule base), and for each host-service combination, we require one source group definition. All components are then present to expand a non-evolvable rule base into a normalized evolvale rule base. Figure 5 visualizes what we want the solution to do.

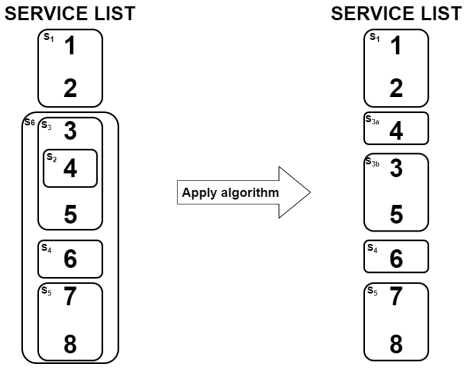

Figure 5. Algorithm objective.

IV. SOLUTION DESIGN

In this section, we will discuss the different components that will make up the algorithm. We start by justifying the choice for Iterated Local Search as metaheuristic [18] [19] [20]. We will discuss the nature of the initial solution, the set of feasible solutions and the objective function associated with a solution. We continue by defining the move type, move strategy, perturbation and stop condition of the Iterated Local Search. The last part of this section discusses the solution encoding and special operations performed in the algorithm, and finally, the presentation of the algorithm.

A. Metaheuristic selection

The objective is to disentangle/reshuffle the service definitions into a set of new service definitions that are disjoint but as large as possible. The simplest solution is to make one service definition per port. But some ports belong together to deliver a service. This logic is somewhere buried in the rule base and service definitions. We may not lose it.

Service definitions containing ports that appear in multiple service definitions must be split in non-overlapping service definitions. The result should be that the degree of overlap (or disjointness) of all service definitions decreases as more service definitions are split. If we measure somehow the degree of disjointness of all the service definitions and see that after a split, the degree of disjointness improved, we know we found a better solution than before.

A Local Search heuristic is a suitable method for organizing such gradual improvement process. To avoid getting stuck in a local optimum (see further), the Local Search will be upgraded to an Iterated Local Search (ILS). The Iteration part should avoid getting stuck in a local optimum where we can no longer perform splits and improve the disjointness. The

iteration part should perform a perturbation, a special kind of split, that will allow the continuation of the search for improvement.

B. Initial Solution and Neighbourhood

The initial solution is given. It is the rule base with all the service definitions. The set of all service definitions is our Neighbourhood. We will have to pick a service definition, check if it is disjoint and if not, split it and see how this affects the solution - disjointness improved or not. The solution space SP for the service definitions consists of all possible combinations of ports. If the number of distrinct ports in the service groups equals P, then the SP is:

$$\mathbf{SP} = \sum_{k=1}^{P} \binom{P}{k} \tag{3}$$

P can be max 2^{17}. We are looking to find a new solution, that is part of the solution space, in which all service definitions are disjoint yet grouped in groups of maximum size.

C. Objective Function

To know if the splitting of a service definition is improving the solution, we need a mechanism to express the degree of disjointness of the service definition. Each definition contains ports and those ports may be part of multiple definitions. We define the port frequency of a port as the number of times this port appears in a service definition. The higher the frequency, the more the need to splitting this port off.

We define the **DI**, the Disnointness Index of a service definition $\mathbf{S_x}$, as the SUM of the port frequencies **PF** of the ports *px* of $\mathbf{S_x}$, divided by the number of ports in $\mathbf{S_x}$.

DI(S_x)=$\frac{\sum_{i=1}^{nx} PF(p_x)}{nx}$ with nx = $\mid S_x \mid$= number of ports in S_x

DI is one if all ports only appear once in a service definition. A DI of one means the service definition is fully disjoint.

We define the Objective Function **OF** as the sum of all **DI** of all service definitions.

OF = $\sum_{i=1}^{n} DI(S_i)$ with n the number of service definitions in the solution.

If the Objective Function value is equal to the number of service definitions, then we have found an optimal solution. Not necessarily a Global Optimum as making service definitions of one port would also yield to an Objective Function equal to the number of service definitions.

D. Feasable Solutions

Whatever kind of splits we will be performing, the original filtering logic of the rule base must be maintained, meaning that splitting service definitions will result in splitting rules to have the identical rule base behaviour. The original rule will have to be removed from the rule base and replaced by a number of rules equal to the splits size (split in 2 groups, 3 groups, etc.).

E. Move Type

The move type will be a split of a service definition. A service definition can:

- be a subset of existing service definitions
- be the superset of existing service definitions
- have a partial overlapping with other service definitions.
- be a combination of the above.

The split during the Local Search will concists of splitting, carving out, all existing subgroups. We call the split the full-carve-out move. This split is chosen as it resolves both the sub and superset case.

Example: A service definition **S**={1,2,3,4,5,6,7}. There also exists service definitions $\mathbf{S_1}$={1,2} and $\mathbf{S_2}$={5,7}. Carving out $\mathbf{S_1}$ and $\mathbf{S_2}$ from **S** gives, $\mathbf{S_1}$= {1,2}, $\mathbf{S_2}$={5,7} and **S'**={3,4,6}

This move type is not able to handle partial overlapping service defintions. It is expected that when all carve outs are done, there will be a number of overlaps remaining that require a diffent kind of move (see furhter).

F. Move Strategy

All services with a **DI** greater than one are candidates for splitting. It seems logical to start splitting the service with the largest **DI**. If that service cannot be split (no subgroups), then the second-largest **DI** is taken, etc. If a group can be split, the impact of the split is calculated. When **OF** improves (=decends), the move is accepted and executed. If not, the next service in the sorted service **DI** list is chosen. The move-strategy is a variant of the First Improvement strategy of the ILS metaheuristic; a variant as we first order the service **DI** list and take the top element from the list.

G. Perturbation

The carve-out of subgroups cannot remove all forms for disjointness. Correlated (partially overlapping) service definitions cannot be split this way. That is why, when no more carve-outs are possible, a new split operator is required. The operator will determine if a service definition overlaps with another service definitions. If it does, the intersection is split-off. By splitting of this intersection, a new service definition will be created that may be inclusively matching with the other service definitions. Another iteration of the Local Search will investigate this and perform the required carve-outs. We consider this kind of split as a perturbation.

H. Stop Conditions

If all possible carve-outs are done, and all perturbations are done, then there are no more inclusively matching and correlated rules. All port frequencies are equal to one, all service group **DI**'s are equal to one, and **OF** will equal the number of service definitions. The solution cannot be improved anymore.

 ISBN: 978-1-61208-850-1

I. Solution Encoding

The algorithm has been implemented in JAVA. The different components of the solution are implemented as JAVA classes. We tried to stay as close as possible to the NS principles by defining data classes, which only contain data and convenience methods to get and set the data, and task classes used to perform actions and calculations on the data objects.

1) Port: Services contain ports. A port is linked to a protocol (TCP or UDP). PortRange is the class representing a range of ports with an associated protocol.

```
public class PortRange
{
private String protocol;
private int begin;
private int end;
}
```

For a single port, begin = end.

2) Port Frequency: Within a solution, each port will have a frequency that is equal to the number of service definitions in which this port appears. PortFrequency is the class representing the port frequency and the service definitions containing that port.

```
public class PortFrequency
{
private int portnumber;
private int frequency;
private ArrayList<String> group_occurancelist;
}
```

3) Port Frequencies list: PortFrequencies class is the list of all ports existing in a solution and for each port the port frequency in the solution. The i^{th} element of the array represents port i. The content of the i^{th} element contains the port frequency information of port i. As there are TCP and UDP ports, two arrays are required for a full port frequency list.

```
public class PortFrequencies
{
private PortFrequency TCP_portfrequency[]=
   new PortFrequency[65536];
private PortFrequency UDP_portfrequency[] =
   new PortFrequency[65536];
}
```

4) Service: The Services class represent a service definition and contains all port ranges, UDP and TCP, associated with the service.

```
public class Service
{
private String name;
private ArrayList<PortRange> udp_ranges;
private ArrayList<PortRange> tcp_ranges;
}
```

5) ServiceList: The ServiceList class is the list of all service definitions of a solution.

```
public class ServiceList
{
private String name;
private ArrayList<Service> servicelistitems;
}
```

6) Service DI: For each service definition, the disjointness index must be calculated and stored. The disjointness index is stored in the Service_DI class.

```
public class Service_DI
{
private Service service;
private double disjointness_index;
}
```

7) ServiceDIList: The ServiceDIList class is a list of all DI's of all service definitions of a solution. This list represents the neighbourhood as this list will be used to iterate over. The service DI list is an ordered list, with the service with the highest DI as the first element of the list.

```
public class Service_DI_List
{
private ArrayList<Service_DI> service_DI_list;
}
```

J. Operations

The algorithm contains a number of tasks that perform actions on and with the data classes. The most important and relevant ones are listed in this section.

1) PortFrequenciesConstructor: The PortFrequenciesConstructore will calculate the port frequencies of all ports used in all services. It takes the current servicelist as input. The result - a PortFrequenciesList - is accessible via a get-method.

2) Service_DI_List_Creator: The Service_DI_List_Creator will calculate the DI of all services. The inputs are the current service list and portfrequencieslist and the result - a ServiceDIList - is accessible via a get-method.

3) Service_Split_Evaluator: The Service_Split_Evaluator will perform a full-carve-out-split. The inputs are the service to split, the current service list, and the current portfrequencieslist. The result of the split, being the a new ServiceList, a new ServiceDIList, a new PortFrequenciesList and the value of the objective function, are accessible via get-methods.

4) Service_Perturbation: The Service_Perturbation will check if a perturbation is possible and if so, perform it. The inputs are the current servicelist and the portfrequencieslist. The results of the split, being the new ServiceList, new ServiceDIList, new PortFrequenciesList and the value of the objective function, are accessible via get-methods.

K. The Iterated Local Search Algorithm

Algorithm 1 (see Figure 6), is the ILS algorihtm designed according to the components described in previous sections. The important variables are:

- sl = the service list.
- pfl = the portfrequencies list.

- sdil = the service DI list.
- of = objective function value of a solution.
- fully_disjoint = boolean indicating if the solution is fully disjoint.
- end_of_neighbourhood = boolean indicating if the full neighbourhood has been seached.
- objective_function_improvement = boolean indicating is the objective function has improved.
- neighbourhoodpointer = index of an element in the sorted neighbourhood
- service_to_split = service of the neighbourhood that will be investigated for splitting.

V. Solution Demonstration

This section starts with describing the platform used to perform the demonstrations, followed by information on the different data sets that are used to run the algorithm and finished with the results of running the algorithm with the three demonstration sets.

A. Demonstration environment

The algorithm is written in JAVA using JAVA SDM 1.8.181, developed in the NetBeans IDE V8.2. The demonstration ran on an MS Surface Pro (5th Gen) Model 1796 i5 - Quad Core @ 2.6 GHz with 8 GB of MEM, running Windows 10.

B. Demonstration sets

1) Demo set: The first data set consists of a manually created list of service definitions. The set contains a lot of exceptions to test the robustness of the algorithm such as services having different names but identical content, services having almost the same name (case differences) but different content, empty service definition, both TCP and UDP ports etc.

2) Engie Tractebel set: Engie Tractebel Engineering delivered the export of a Palo Alto Firewall used in an Azian branch. The set is a realistic representation of a firewall that interconnects a branch office with the rest of the company network.

3) Engie IT data center set: Engie IT delivered the export of a Palo Alto Firewall used in the Belgium Data centre. The firewall connects different client zones in the data centre with a data centre zone containing monitoring and infrastructure management systems.

C. Demonstration results

The demonstration results show the evolution of 3 indicators. The objective function is the main indicator. Also visualized are the level 1 and level 2 iteration indicators. The level 1 indicator is the number of times that the outer DO loop of the algorithm has run. The indicator measures the number of times a perturbation is made. The level 2 indicator is the number of times the inner DO loop of the algorithm runs within a given level 1 iterations. This means that each time a new level 1 iteration runs, the level 2 iterator is reset. The X-axes of the represents the cumulative level 2 iterations.

Algorithm 1: ILS for service list normalization

```
sl = load_initial_solution(filename);
pfl = portfrequen-
  cies_list_constructor(sl).get_portfrequencies_list();
sdil = service_di_list_creator(sl, pfl).get_service_di_list();
of = service_di_list.get_objective_function();
fully_disjoint = FALSE;
end_of_neighbourhood = FALSE;
objective_function_improvement = FALSE;
while NOT full_disjoint AND NOT end_of_neighbourhood
 do
    neighbourhood = sdil.sort;
    neighbourhood_pointer = 1 (top of list)
    objective_function_improvement = FALSE;
    while NOT improvement_objective_function AND NOT
     fully_disjoint AND NOT end_of_neighbourhood do
        servicer_to_split = neighbour-
          hood.get_element(neighbourhood_pointer);
        service_split_evaluator(service_to_split, sdil, pfl);
        objective_function_improvement = ser-
          vice_split_evaluator.get_objective_function_improved();
        if objective_function_improvement = TRUE then
            sl= service_split_evaluator.get_service_list();
            pfl = ser-
              vice_split_evaluator.get_portfrequencies_list();
            sdil =
              service_split_evaluator.get_service_di_list();
            fully_disjoint =
              service_di_list.is_fully_disjoint_check();
        else
            neighbourhood_pointer ++
        end
        end_of_neighbourhood =
          sdil.end_of_list_check(nieghbourhood_pointer);
    end
    if end_of_neighbourhood then
        service_perturbation_exists =
          service_perturbation.perturbation exists(sl,pfl);
        if service_perturbation_exists then
            sl= service_split_evaluator.get_service_list();
            pfl = ser-
              vice_split_evaluator.get_portfrequencies_list();
            sdil =
              service_split_evaluator.get_service_di_list();
            fully_disjoint =
              service_di_list.is_fully_disjoint_check();
            end_of_neighbourhood = FALSE;
        else
            end_of_eighbourhood = TRUE;
        end
    end
end
if fully_disjoint then
    PRINT "Probably the Global Optimum has been
      found";
else
    PRINT "Local Optimum found";
end
PRINT "Solution = " + sl.get_overview();
```

Figure 6. ILS based algorithm

1) Demo set: The algorithm produces its result in 1 to 2 sec. The start value of the objective function is 110, and the end value is 34. The total number of level 1 iterations is 43, and the total number of level 2 iterations is 568. The algorithm starts 28 service definitions. The algorithm ends with 34 service definitions. The objective function goes down in an almost exponential mode. The Level 2 Iterations go up in an almost logarithmic mode, and the Level 2 Iterations follow a kind of saw-tooth function, with a frequency that goes towards the size of the neighbourhood. Figure 7 shows the evolution of the OF, level 1 and level 2 iterations during the exectution of the algorithm.

2) Engie Tractebel set: The algorithm produces its result in 190 sec. The start value of the objective function is 278, and the end value is 62. The total number of level 1 iterations is 23, and the total number of level 2 iterations is 358. The algorithm starts 79 service definitions. The algorithm ends with 62 service definitions. The objective function goes down in staged mode. The Level 2 Iterations go up in an almost logarithmic mode, and the Level 2 Iterations follow a kind of saw-tooth function, with a frequency that goes towards the size of the neighbourhood. Figure 8 shows the evolution of the OF, level 1 and level 2 iterations during the exectution of the algorithm.

3) Engie IT data center set: The algorithm produces its result in 360 sec. The start value of the objective function is 3876 and the end value is 418 . The total number of level 1 iterations is 127 and the total number of level 2 iterations is 10.835. The arlgorithm starts 459 service definitions. The algorithm ends with 418 service definitions. The objective function goes down in staged mode. The Level 2 Iterations go up in an almost logarithmic mode and the Level 2 Iterations follow a kind of saw-tooth function, with a frequency that goes towards the size of the neighbourbood. Figure 9 shows the evolution of the OF, level 1 and level 2 iterations during the exectution of the algorithm.

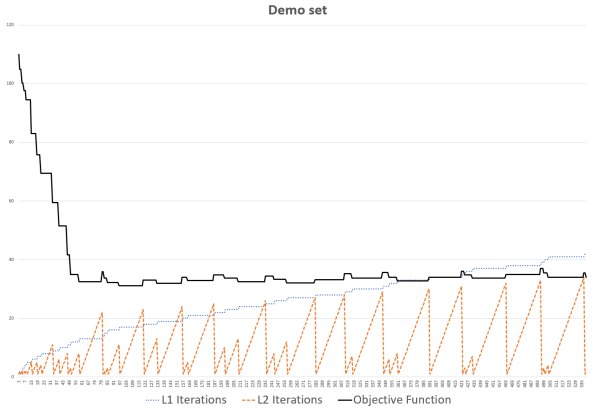

Figure 7. Objective Function, L1 Iteration and L2 Iteration for the demo set.

VI. Solution Evaluation and Discussion

In this section, we will evaluate and discuss the algorithm, starting with the Big O of the algorithm. We continue to specify the impact of the splits on the rule base and by positioning the algorithm as an essential building block in the conversion of a rule base into an evolvable rule base. We conclude by proposing some potential performance enhancement methods and an alternative for the algorithm.

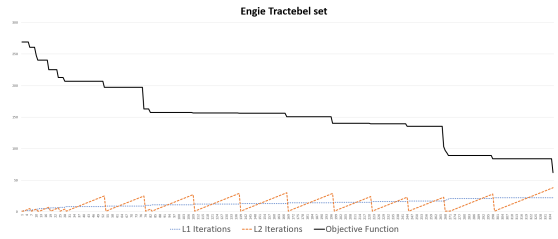

Figure 8. Objective Function, L1 Iteration and L2 Iteration for the Engie Tractebel set.

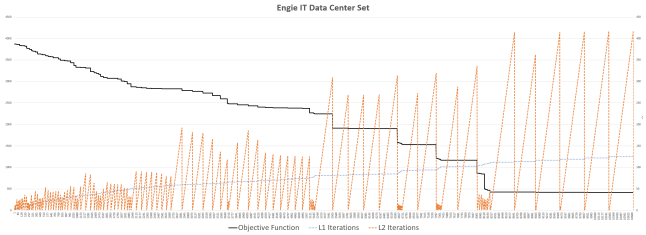

Figure 9. Objective Function, L1 Iteration and L2 Iteration for the Engie IT data center set.

A. Big O of the algorithm

The algorithm contains two nested loops that both can iterate over the full neighbourhood, meaning the algorithm will be quadratic with respect to the size of the neighbourhood. The operations performed in the most inner loop, like Service_DI_list_Creator, Servicer_split_Evalturator are also proportional to the size of the neighbourhood. We can thus conclude that the Big O of the complete algorithm is cubic - $O = n^3$, where n is the size of the neighbourhood (= size of the solution = the number of service definitions)

B. Impact of the splits on the rule base

Each time a service is split, there is an impact on the rule base. All rules containing this service must be adjusted according to the result of the split. Two kinds of adjustments are required.

- Split rules: The rules containing this service must be split in 2 rules that contain the results of the split.
  Example:
  Before split: Rx = source-destination-service
  Split: service splits into service1 and service2
  After split: Rx1 = source-destination-service1 and Rx2 = source-destination-service2

- Rename services: When service splits result in existing services, those services will be renamed to track the changes. All rules that are impacted by this rename must be adjusted. Example:
  Before split: servicex
  Split: service x splits in service x' and service x", but those existed already under the names service xV5 and service yV8. Service xV5 becomes service xV6,

 ISBN: 978-1-61208-850-1

and service yV8 becomes service yV9.
After split: service x is replaced by service xV6 and service yV9. Service xV5 is replaced by service xV6 and service yV8 is replaced by service yV9.

The algorithm does not include the adjustments of the rules, but counts the number of times such an adjustment is required. Table I shows the different test sets, the initial and final value of the objective function, while Table II shows, for the different test sets, the initial rule base size and the number of additional rules due to the splits. Further work is required to adjust the algorithm to perform the actual splits and to have a better view on the actual amount of additional rules.

TABLE I. EVOLUTION OBJECTIVE FUNCTION

Test set	Inital OF	Final OF
Demo set	110	34
Engie Tractebel set	278	62
Engie IT data center set	3874	418

TABLE II. IMPACT ON THE RULE BASE

Test set	Initial size rule base	Extra rules	Service renames
Demo set	NA	74	55
Engie Tractebel set	37	135	152
Engie IT data center set	522	2940	2976

C. Building block for evolvable rule base

The list of disjoint services is the essential building block for building an evolvable rule base. According to artifact of Section III-A, for each service, there should be as many destination groups created as there are hosts offering this service. And for each destination group created in this manner, there should be one source group created. The population of those destination and source group can happen via investigation of the existing rule base.

D. Impact on the size of the rule base

The algorithm has been demonstrated in only 3 test cases. More test cases are required to get a better insight into the impact of splitting services into disjoint servers on the rule base's size. A more detailed study of different firewall types within Engie is on the researcher's agenda.

E. Potential performance improvements

1) Pre-processing: The firewall configuration contains both service definitions and service group definitions. Service groups aggregate service definitions. In the simulations, service groups are part of the service list, and logically those are the first that will undergo the full-carve-out operations. It could be beneficial to exclude service groups. This would require the replacement of the service groups used in the rule base by their individual services and splitting of rules accordingly. This pre-processing step also takes time, and it remains to be seen if it improves performance.

2) Memory: The algorithm would benefit from some memory as defined in metaheuristics. All groups that are disjoint no longer require checking if they are disjoint and can be removed from the search list. This could reduce the size of the neighbourhood dynamically and improve performance.

3) Deterministic approach: Tests of the algorithm show that there is always converges to the same solution for a given initial solution. Although we cannot prove it formally (yet), for a given initial solution, there is convergence to one solution that seems to be the Global Optimum. The creation of the algorithm resulted in a progressive insight about how to disentangle the service definitions. We now believe that the disentanglement can be achieved without the the calculation of the objective function, which is basically saying we no longer have an Iterated Local Search algorithm but an algorithm that will follow a predetermined path toward the solution.

VII. CONCLUSION

Using Iterated Local Search, an algorithm was created that allowed the disentanglement of a set of groups that are nested and overlapping, into a set of groups that is disjoint from each other. Such an algorithm can be applied in the specific context of making firewall rule bases evolvable. The algorithm has been demonstrated successfully. Progressing insight during the creation of the algorithm points toward a deterministic algorithm. More firewall exports are required to get a better idea on the impact of the splitting of services into disjoint services, on the size of the rule base.

REFERENCES

[1] H. Shel and A. Spiliotes, "The State of Network Security: 2017 to 2018", Forrester Research, November 2017

[2] "2018 State of the firewall", Firemon whitepaper, URL https://www.firemon.com/resources/, [retrieved: April, 2021]

[3] "Firewall Management - 5 challenges every company must address", Algosec whitepaper, URL https://www.algosec.com/resources/, [retrieved: April, 2021]

[4] G. Haerens and H. Mannaert, "Investigating the Creation of an Evolvable Firewall Rule Base and Guidance for Network Firewall Architecture, using the Normalized Systems Theory", International Journal on Advances in Security, Volume 13 nr. 1&2, pp. 1-16, 2020

[5] H. Mannaert, J. Verelst and P. De Bruyn, "Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design", ISBN 978-90-77160-09-1, 2016

[6] E. Al-Shaer and H. Hamed, "Taxonomy of conflicts in network security policies", IEEE Communications Magazine, 44(3), pp. 134-141, March 2006

[7] E. Al-Shaer, H. Hamed, R. Boutaba and M. Hasan, "Conflict classification and analysis of distributed firewall policies", IEEE Journal on Selected Areas in Communications (JSAC), 23(10), pp. 2069-2084, October 2005

[8] M. Abedin, S.Nessa, L. Khan and B. Thuraisingham, "Detection and Resolution of Anomalies in Firewall Policy Rules", Proceedings of the IFIP Annual Conference Data and Applications Security and Privacy, pp. 15–29, 2006

[9] E. Al-Shaer and H. Hamed, "Design and Implementation of firewall policy advisor tools", Technical Report CTI - techrep0801, School of Computer Science Telecommunications and Information Systems, DePaul University, August 2002

[10] S. Hinrichs, "Policy-based management: Bridging the gap", Proceedings of the 15th Annual Computer Security Applications Conference, pp. 209-218, December 1999

 ISBN: 978-1-61208-850-1

[11] A. R. Hevner, S. T. March, J. Park and S. Ram, “Design Science in Information Systems Research”, MIS Quarterly, Volume 38, Issue 1, pp. 75-105, 2004

[12] P. Johannesson and E. Perjons, “An Introduction to Design Science”, ISBN 9783319106311, 2014

[13] W. R. Stevens, “TCP/IP Illustrated - Volume 1 - the Protocols”, Addison-Wesley Publishing Company, ISBN 0-201-63346-9, 1994

[14] H. Zimmermann and J. D. Day, “The OSI reference model”, Proceedings of the IEEE, Volume 71, Issue 12, pp. 1334-1340, Dec 1983

[15] H. Mannaert, J. Verelst and K. Ven, “The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability”, Science of Computer Programming, Volume 76, Issue 12, pp. 1210-1222, 2011

[16] P. Huysmans, G. Oorts, P. De Bruyn, H. Mannaert and J. Verelst, “Positioning the normalized systems theory in a design theory framework”, Lecture notes in business information processing, ISSN 1865-1348-142, pp. 43-63, 2013

[17] G. Haerens, “Investigating the Applicability of the Normalized Systems Theory on IT Infrastructure Systems, Enterprise and Organizational Modeling and Simulation”, 14th International workshop (EOMAS) 2018, pp. 23-137, June 2018

[18] M. Rafael, P. M. Pardalos and M. G. C. Resende, “Handbook of Heuristics”, ISBN 978-3-319-07123-7 IS, 2018

[19] Z. Michalewicz and D. B. Fogel, “How to Solve It: Modern Heuristics”, ISBN 978-3-642-06134-9, 2004

[20] E. Talbi, “Metaheuristics - From Design to Implementation”, ISBN 978-0-470-27858-1, 2009

 ISBN: 978-1-61208-850-1