Industrial Grade Methodology for Firewall Simulation and Requirements Verification

Ramon Barakat, Faruk Catal, Nikolay Tcholtchev, Yacine Rebahi and Ina Schieferdecker
Fraunhofer Institute for Open Communication Systems (FOKUS), Berlin, Germany
{firstname.lastname}@fokus.fraunhofer.de

***Abstract*—Firewalls are a critical part in any security framework.Most firewalls consist of a large amount of sequential rules that are unstructured and confusing. Unfortunately, because a lot of rules configuration work is done manually by the network administrators, misconfigurations are very common and can affect the reliability of the firewall. Identifying such anomalies is a challenging task. In this paper, we propose a tree based simulation and verification model to verify if the implemented firewall of a system is in compliance with the belonging firewall requirements. The proposed methodology was developed in relation with the H2020 FORTIKA project and was evaluated in the scope of case studies with industrial partners. The case studies in question related to large scale telecom infrastructures involving critical scenarios in the scope of Smart Cities in general and SME cyber-security protection. Thereby, the executed case studies demonstrate how our approach can lead to improved structuring of firewalls and belonging rules, to the comfortable visualization of firewall structures and decision patterns, and finally to the verification of system and context requirements imposed by the firewall operation environment.**

***Index Terms*—firewall, verification, simulation, requirements traceability, quality assurance, model-checking, model testing.**

I. INTRODUCTION

It is a common practice to implement a firewall based on the configuration of a set of sequential rules which are triggered for each packet or (tracked) connection. Based on packet fields such as *protocol type*, *source IP address* or *port number* a rule decides whether to accept or to decline a packet or to continue the processing with a following rule. Depending on the complexity of the system and the number of requirements, the amount of sequential rules becomes large and difficult to comprehend very quickly. Therefore, the firewall configuration is hard to understand. Possible security problems, leaks and firewall misconfigurations are very difficult to analyse and detect. Misconfigurations inside the firewall can block legitimate packages or even worse, lead to potential security gaps that can have fatal consequences in safety-critical systems.

To ensure that the configured rule set fulfills the requirements is a very important but challenging task. Unfortunately the established firewall tests are usually not sufficient, mostly for combinatorial reasons since it is impossible to test all combinations of parameters for the packets and connections to be blocked. Even with methods like fuzzing [1] it is not possible to completely explore the input space for firewalls given the large amount of invalid test data that needs to be generated, in order to verify the proper firewall functioning according to relevant system requirements.

Our approach is to extract an abstract representation of the implemented firewall of a system. We developed an algorithm that can reverse engineer an abstract decision tree from the belonging firewall implementation. This tree provides a better understandable overview and transparency relating to the implemented firewall. Additionally, this tree can be used for simulation and various model checking and verification procedures.In this line of thought, our methodology uses the abstract decision tree to verify if the firewall satisfies the operational and functional requirements. If a requirement is not satisfied, the simulation and model-checking facilities provide and describe a counterexample showing the violation of the specific requirement. For the simulation and verification, we are using the Uppaal [2] model-checker. However, any other tree or automata[1] based model-checking tool can be used.

The proposed verification methodology was developed within the FORTIKA project. The FORTIKA project [3] funded by the European Union's Horizon 2020 Framework Program for Research and Innovation works on effective cyber-security solutions for trusted SMEs IT Ecosystem. The firewall verification methodology proposed in this paper was applied in a project with large scale telecommunication industrial partners. Within the project, the Internet-connected system that should be checked uses *nftables* [4] to configure the firewall and had to be verified against more than 150 requirements.

II. RELATED WORK

For the firewall design several high-level firewall languages like *FLIP* [5], *Firmato* [6] or the Abstract Firewall Policy Language (AFPL) [7] have been developed. In the industry however, these are used only rarely so far [7]. High-level firewall languages are helpful, but they are still rule-based and do not avoid misconfigurations at all. Gouda and Liu [8] propose a method for the structured firewall design by using a *Firewall Decision Diagram (FDD)* to construct a complete, conflict-free and compact sequence of firewall rules. Furthermore, Liu presents in [9] a tool that verifies whether a property is satisfied by the firewall policy by using decision diagrams. To verify the designed firewall, the query concept introduced in [10] is used. Approaches to identify anomalies

[1]A tree can be seen as a special form of an automata.

in a policy are proposed is [11], [12], [13]. However, these verification methods are only examining the firewall policies and not the implemented firewall.

The most common method to identify errors in an implemented firewall is the firewall testing. Thereby, different packets that do or do not satisfy the firewall policy are fired to the firewall to check if they will be processed correctly. In [14] a specification-based method is proposed to derive test cases for firewall testing. An automated testing paradigm is proposed by [15]. Regression and conformance testing for firewalls are described in [16], [17], [18]. However, testing in general is not able to ensure that the system fulfills the predefined firewall policy and goals. Because of the complexity and large input space, a suitable amount of test cases is hard to define. Elmallah and Gouda are presenting in [19] that the firewall verification problem is NP-hard, whilst testing alone is by far not enough to satisfy the increased cyber-security requirements for critical infrastructures.

In our approach, we extract an abstract representation of the firewall to reduce the complexity and to be able to execute model-checking algorithms. The firewall policy and requirements are used to derive the queries that should be executed against the extracted firewall model, in order to verify its integrity with model-checking techniques.

[20] proposes an approach for extracting state automata from firewall policy rules based on an intermediate step of representing the firewall design in a table form. Subsequently, the automata is analyzed and various types of quality issues recognized and removed (e.g. redundancy constructs). [21] demonstrates a technique for the analysis of *iptables* based firewalls within the Isabelle/HOL theorem prover. In this line of tought, a presumed formal semantics for the behaviour of firewalls is used to define proper conditions and logical statements witin the theorem prover and to iteratively improve the general quality and readability of a firewall implementation. [22] provides a comprehensive recent survey and analysis of various IPsec/firewall policies and discusses on possible quality improvement options and security policy verification approaches.

During the past years, the protection of network nodes and infrastructures has been increasingly realized through different network function automations realizing the notions of Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS). For example, the domain of autonomic networking has come up with various efforts and architectures to automatically monitor network traffic and issue corresponding intrusion detection/prevention actions when required. Thereby, [23] [24] [25] constitute such example research and development efforts from autonomic networking, which are defined in the scope of an autonomic networking architecture being at the heart of related ETSI standardization efforts [26]. Beyond the autonomics domain, the intrusion detection/prevention through the usage of Virtual Network Functions (VNF) is a hot topic nowadays. Thereby, virtualized network nodes and functions are distributed across the network infrastructure and equipped with the belonging intelligence and logic as to automatically and collaboratively monitor, identify and block network attacks and anomalies [27] [28]. [29] proposes a framework for the trusted and authorized distribution of short-lived VNF based firewalls in a distributed network infrastructure. The framework facilitates the integrity of the network infrastructure based on a blockchain ledger, allowing to verify and track the quality of the provided short-lived VNF, e.g. based on the vendor/provider of the belonging functions and his reputation/record.

III. Context Analysis

It is very often the case in industrial settings that firewall implementations are put in place in a straight manner, i.e. existing *iptables* [30] or *nftables* [4] scripts are installed on a device and are correspondingly being adapted to the specific security situation. Furthermore, network experts tend to kick start a firewall project by directly coding/configuring the firewall rules (e.g. *iptables*, *nftables*, ...). Hence, it is hard to determine the amount of requirements coverage because there is no link between a requirement and the corresponding firewall rule. If the system has to be approved by a certification authority, then the firewall behaviour should be depicted in a comprehensible way, which is hard to achieve given the large amount of rules and belonging configurations.

An unstructured approach, combined with the lack of proper requirements management, often leads to a situation where a firewall is already in place but cannot be fully trusted – especially in critical infrastructures – provided the lack of clear traceability between requirements and firewall artefacts, as well as the combinatorial difficulties to fully test a firewall implementation.

IV. Verification Methodology

To overcome the issue described in the previous section, a model-checking approach can be very helpful. An abstract reverse engineered model of the implemented firewall supports the transparency and enables the verification of the firewall against the system requirements. Fig. 1 describes the steps required for achieving the formal model-based verification of a firewall.

A. Reverse Engineered Abstract Decision Tree

Provided that a set of implemented firewall scripts exists (*Implemented Firewall* at the top in Fig. 1), then various reverse engineering scripts (e.g. Python based) can parse the grammar of the firewall implementation (e.g. *iptables*, *nftables* or *netfilter*), extract the logic behind and transfer it to an abstract decision tree. This tree represents the firewall design architecture in a more structured way. Also the system and packet properties (e.g. *IP-ranges*, *interfaces*, *protocols* and so on) considered by the firewall can be derived and transferred to individual structural templates that will be used to configure and simulate certain conditions of the system.

The resulting trees can be stored (as XML) and thus can be prepared for a model-checking environment such as the Uppaal [2] model-checker. The corresponding Uppaal systems - i.e.the

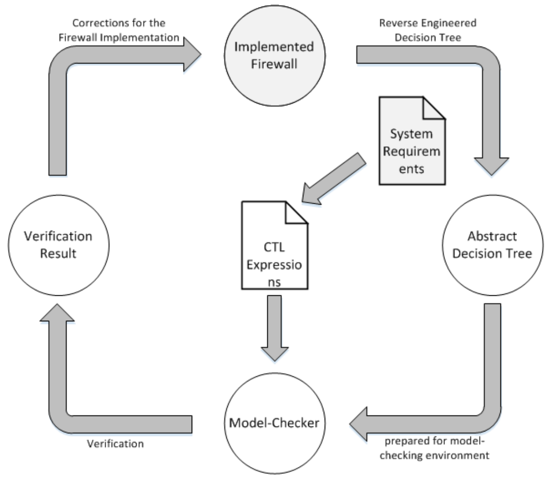


Fig. 1. Overview of the Process leading to Firewall Verification and Simulation

XML file with corresponding context and variable definitions - are automatically generated and loaded into the model-checking environment. Thanks to the structural templates, different system states and network packages can be simulated. The ability to simulate the firewall significantly improves the transparency, requirements traceability and quality assurance relating to the implemented firewall.

B. Requirement Constraints

From the *System Requirements* (upper right corner in Fig. 1) a set of logical constraints is generated semi-automatically. These logical constraints are described in CTL (computation tree logic) [31] - a query language used for model-checking in Uppaal[2]. CTL is a widely used logic for formulating model queries in the scope of tree based model-checking frameworks and tools. In our case, the CTL queries are derived automatically from the system requirements imposed on the firewall implementation in terms of traffic matrix and access control rules. This matrix specifies for each requirement if a package with the given properties in combination with the specific system settings - that are also given in the matrix - should be accepted or dropped.

C. Verification Process

The Abstract Decision Tree (ADT) and the requirements based CTL constraints can be given to a model-checker (at the bottom in Fig. 1) like Uppaal and a model-checking procedure can be started, in order to verify and examine whether the systems requirements - being continuously updated - are fulfilled. The model-checker checks for each path of the abstract decision tree and each possible configuration (except for those that are specified by the requirement itself) if the CTL expression holds or not. If not, the expression - and so the system requirement - is not satisfied. The firewall implementation can be corrected and verified again until all requirements are satisfied.

[2]Uppaal uses a subset of CTL.

V. Abstract Decision Tree

An exemplary firewall implementation that uses *nftables* and illustrates the rules and jumps during the packet processing chain can be found in Listing 1. Based on the packet properties that will be monitored by the firewall, a so called *packet context* can be defined. A *packet context* $\mathcal{P}$ is a n-tuple of packet properties examined by the firewall. The *packet context* can look, for example, like $\mathcal{P}_{ex}$ that contains the source IP address (*srcIP*), the destination IP address (*dstIP*), the *port* and the *protocol* of a data packet.

$$\mathcal{P}_{ex} = (srcIp, dstIP, port, protocol)$$

```
table ip filter {
  chain IN {
    ip saddr @VPN_SRV_IP goto IN_SRC_VPN
    ip saddr @OWN_IP goto IN_SRC_OWN
    goto IN_SRC_ELSE
  }

  chain IN_SRC_VPN {
    ip protocol ipsec goto IN_SRC_VPN_ACC
    goto IN_SRC_VPN_PROTO_ELSE
  }

  chain IN_VPN_ACC {
    # ACCEPT Packet
    accept
  }
  ...
```

Listing 1. Exemplary *nftables* Implementation

The resulting ADT is referred *abstract* because of the abstraction of the processed network packets. For instance, instead of using concrete IP-Addresses, generic variables will be used that indicate the segment the IP is coming from, or the general (physical) network which relates to corresponding network/system interfaces. Therefore, the destination IP of a packet will be, for example, *LOCAL_HOST_IP* instead of 192.168.x.x . Similarly to the *packet context* a *system context* $\mathcal{S}$ can be defined that specifies the state of key system parameters (e.g. if a VPN connection is enabled).

In this paper, the ADT is defined as follows: An abstract decision tree $\mathcal{T} = (\mathcal{V}, \mathcal{E}, \mathcal{C})$ is a triple with a finite and non-empty set of nodes $\mathcal{V}$, a set of conditions $\mathcal{C} : \mathcal{P} \cup \mathcal{S} \to$ *bool* and a set of transitions $\mathcal{E} \subset \mathcal{V}$ x $\mathcal{C}$ x $\mathcal{V}$ for which the following holds:

1) There exists a unique element $r \in \mathcal{V}$ (called *root*) such that $\nexists v \in \mathcal{V},\ (v, r) \in \mathcal{E}$
2) $\forall v \in \mathcal{V}$, there is exactly one path from the *root* node to node v.

A *path* to a node x_n is a set of transitions with $path(x_n) = \{(x_0, c_1, x_1), (x_1, c_2, x_2), ..., (x_{n-1}, c_n, x_n) \mid (x_{i-1}, c_i, x_i) \in \mathcal{E} \ \wedge \ \forall c_i,\ c_i$ is satisfied $\wedge\ x_0 = r\}$.

The length $n = |path(x_n)|$ of a path is the number of transitions in the path. A *level* of the tree $\mathcal{T}$ is the set of nodes with the same path length. Node $y \in child(x)$ is called child of x if $(x, c, y) \in \mathcal{E}$. We will call each node with at least one child a *decision node*. A node without a child is called a *leaf*. The ADT representing a firewall implementation should satisfy the following properties:

(a) Given $r \in \mathcal{V}$ as the *root* node of the ADT, then we specify $child(r) = \{IN, OUT, FWD\}$ where IN is the "root node" of the subtree for the incoming traffic, OUT is the "root node" of the subtree for the outgoing traffic and FWD is the "root node" of the subtree for traffic that should be forwarded[3].

(b) Each *level* of a subtree checks a specific property of $\mathcal{P}$ or $\mathcal{S}$.

(c) $\forall(x, c_1, y_1) \in \mathcal{E}, \forall(x, c_2, y_2) \in \mathcal{E}, ctx \in \mathcal{P} \cup \mathcal{S}$ with $y_1 \neq y_2$, $c_1 \neq c_2$, $c_1(ctx) \Rightarrow \neg\, c_2(ctx)$
It means that for each *decision node* there is at most one reachable child according to a particular system state and packet context.

(d) The set of leaves can be separated in accepting and non-accepting leaves.

Fig. 2 shows a section of such an ADT resulting from the reversed engineered *nftable* implementation from Listing 1 in Uppaal. The subtrees mentioned in (a) provide an initial distinction of the traffic to check. Property (b) describes the structure of the ADT. In this way, the order of packet and system properties to check can be defined. By a suitable arrangement of this order, the state space can be reduced and the verification can be accelerated. This arrangement mostly depends on the system and type of traffic. For some systems it makes sense to check the source IP address on a higher and the protocol of the package on a lower level in the ADT. But for a system that only accepts a specific protocol, it makes more sense to check the protocol type first. Hence, each subtree of (a) can have a different order as well. Fig. 2 shows that, for example the first level of the input subtree (*IN_L1*) can be checking the source IP address, whereas the first level of the output subtree (*OUT_L1*) would be checking the VPN state. Moreover, property (c) ensures the unambiguity of the paths where the separation in (d) is needed to decide how to handle the packet within the firewall. If the simulation reaches an accepting leaf (e.g. node *IN_VPN_ACC*), then the given packet will be accepted by the firewall. If the simulation reaches a non-accepting leaf (e.g. node *IN_SRC_OWN*) or gets stuck in a decision node (e.g. a packet with protocol type *UDP* in node *IN_SRC_ELSE*), then the packet will be dropped.

VI. Queries Generation and Verification

As described in the previous section, it is required to trace and verify that relevant requirements are indeed fulfilled by a firewall implementation (be it an *iptables* or *nftables* one). Two examples of such requirements are given as follows:

REQ 1:	*Incoming packets should not have the IP address of the system as the source IP address.*
REQ 2:	*If VPN is enabled the system shall allow to send packets with destination IP is VPN_SRV_IP.*

[3]Of course, the amount of subtrees given here may vary depending on the specifics of the firewall system.

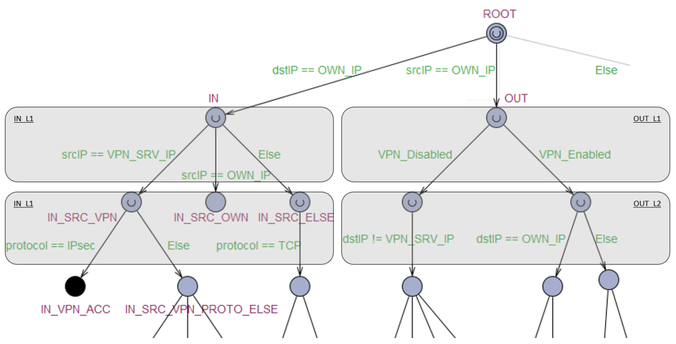


Fig. 2. Exemplary Section of an ADT

Thereby, it is important to formalize the requirement as a set of parameters and values to be set in the *packet context*, reflecting the various environmental variables with their values in the course of firewall packet processing. Having captured these aspects (e.g. in an office packet table handling software), it is easy to generate verification constraints for a model-checker such as Uppaal [2]. Here, it is important to mention that not every single property has to be specified. Only those which are covered by the specific requirement need to be handled with their concrete values. For properties that are not specified in the matrix (see TABLE I for an example) the model-checker will examine each value that can be configured. For the two requirements mentioned above, the belonging specification matrix can be viewed in TABLE I.

TABLE I
Exemplary Requirements Matrix

ID	Accept	Package context			System settings
		srcIP	dstIP	...	VPN
REQ 1	false	not(OWN_IP)			
REQ 2	true		VPN_SRV_IP		enabled

The following shows the CTL[4] representation of the above requirements in a form which can be directly loaded in Uppaal and checked against the firewall model extracted from the *nftables/iptables* implementation within the device in question.

(1) *REQ 1: A[] not(srcIP.OWN_IP & dstIP.OWN_IP & Firewall.accept)*

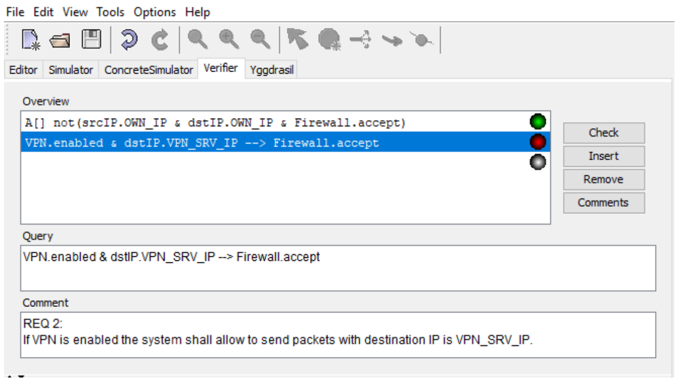
(2) *REQ 2: VPN.enabled & dstIP.VPN_SRV_IP* $\longrightarrow$ *Firewall.accept*

The first expression (1) can be read as follows: On all paths of the tree there should never be a state where the *source IP* and the *destination IP* address is *OWN_IP* and the firewall accepts the packet. Accordingly, expression (2) can be described as: If there is a configuration C where VPN is enabled and *destination IP* is *VPN_SRV_IP*, then on any path of the decision tree a state should be reached where the firewall accepts the packet. Thanks to an initialization phase,

[4]Computation Tree Logic

we ensure that a configuration C stays unchanged during the simulation/examination/verification.

Based on such checking of requirements against the firewall decision tree, it is possible to ensure that all requirements are addressed within the selected firewall design architecture, and that way to establish traceability between the firewall architecture and system requirements accumulated over time. Fig. 3 shows the user interface for model checking in Uppaal with both requirement queries loaded and examined. Here REQ 1 is satisfied (green bullet) and REQ 2 is not satisfied (red bullet).



```
A[] not(srcIP.OWN_IP & dstIP.OWN_IP & Firewall.accept)
VPN.enabled & dstIP.VPN_SRV_IP --> Firewall.accept
```

VPN.enabled & dstIP.VPN_SRV_IP --> Firewall.accept

REQ 2:
If VPN is enabled the system shall allow to send packets with destination IP is VPN_SRV_IP.

Fig. 3. User Interface of the Uppaal Verifier

The Uppaal verification engine can be either executed in the Uppaal GUI (see Fig. 3) or started on the command line. Moreover, each CTL expression (i.e. each system requirement) is verified separately. This gives us the possibility to distribute the verification on different powerful machines and integrate the firewall model-checking process in different DevOps architectures.

VII. Verification Results and Measurements

The following diagrams show the measurements of the verification process for a set of requirements within an industrial case study. These measurements provide indications regarding the operational settings and parameters of the proposed methodology. We deliberately relate to an early work-in-progress version of the verification process with a firewall tree which is far from accomplished, in order to illustrate the overall iterative process. This implies the various results observed and depicted later on in the discussion and leads to clarification regarding the iterative procedures.

Based on the firewall implemented in the industrial case study in question, the individual structural templates shown in TABLE II have been extracted. As a *structural template* we denote a basic structure relating to a parameter or variable in the processing of a packet (or tracked connection) through the firewall. This parameter/variable steers the decisions at different levels in the ADT and basically resembles the conditions from the formal firewall tree definition presented before. For example, in the current case study there are six boolean system settings (first line of TABLE II) which can take one of two values (*Enabled* or *Disabled*). A further example can be illustrated based on the *IP-Address Template* that is used twice in the case study - once for the source and once for the destination IP of a packet - and can take 24 different values which relate to the different network segments involved in the network scenario in question.

TABLE II
Individual Structural Templates

Usage Frequency	Structure	Number of Possible Values
6	Boolean Template	2
2	IP-Address Template	24
2	Interface Template	6
1	Protocol Template	10
3	Others Templates	3

The decision tree itself - resulting from the case study - consist of about 260 nodes separated in two subtrees (*IN* and *OUT*). The verification has been executed on the command line of a virtual machine with the following technical parameters: Intel(R) Xeon(R) CPU E5-2680, 2,4 GHz, 4 cores and 32 GB RAM. It is important to mention that although the virtual machine provides four cores, Uppaal is running with only one core at full capacity. Hence, additional optimization may be achieved by employing more sophisticated versions of Uppaal if available in the future.

In order to examine the operational parameters of the proposed methodology, 80 requirements with their belonging CTL queries have been evaluated. The results of the verification can be found in Fig. 4. As we can observe, a large number of rules (47%) fail to conclude due to memory issues, whilst other 34% find mistakes in the firewall design and end with a *NOT Satisfied* verdict. Indeed, the illustrated situation depicts a typical work-in-progress state towards the refinement of a firewall design within our iterative methodology as depicted in Figure 1. The *NOT Satisfied* CTL rules will be communicated to the firewall developers and an analysis of the problems will be started. The rules which cannot run due to memory issues are further processed in two different ways - either (1) a firewall redesign is considered at crucial structural point that allows these rules to execute within reasonable resource setting, or (2) the rules have to be split in multiple rules of more simple structure that can run against the ADT. The above described tasks and processes run iteratively in close collaboration with the firewall developers until a verifiable firewall tree is achieved, which can be traced and quality assured with regard to fulfilling the identified system requirements.

Coming back to Fig. 5, we can observe the different times required for the execution of the CTL rules with different outcomes. Thereby, the curves relating to the consequent execution are placed on top of each other, in order to give a better feeling regarding the experienced verification time scales. We observe that in general rules ending with a memory issue require much significantly more time than the rules, which end with a defined outcome (be it positive or negative). This stems from the fact that the solver experiences a space explosion

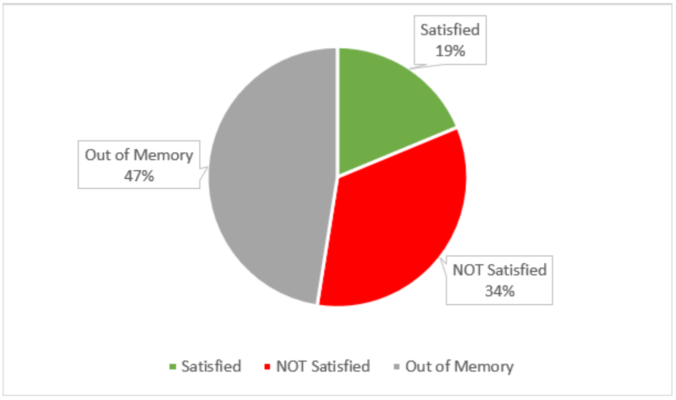


Fig. 4. Verification Results of the Experimental Requirements/CTL Set

and has to load more and more states, which takes larger amount of time, until finally running out of allocated memory. Regarding the rules with positive and negative outcome, we observe that the verification times are of similar magnitude, which is expected given the ability of the solver to reach a conclusion based on the underlying firewall tree structure.

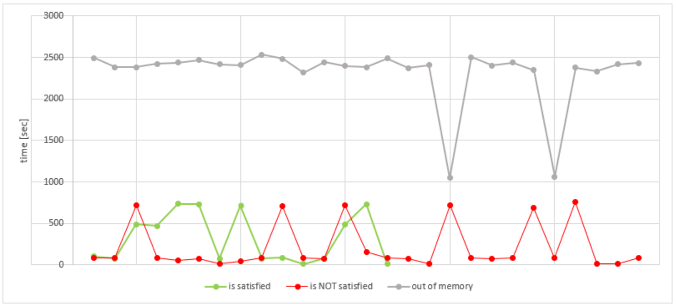


Fig. 5. Verification Time Comparison for the Different Outcomes of the Experimental Requirements/CTL Set

Finally, Fig. 6 and Fig. 7 give further indications for the performance of the verification procedure. Fig. 6 shows the number of states that have been loaded (in memory) for processing during the verification of the belonging CTL rules, while Fig. 7 shows the corresponding throughput in states per second. With some small exceptions the throughput is constantly between 350000 and 400000 states per second. The two figures characterize the operations of the model-checker within the large tree that we examined in our scenario.

A. Verification Problems

Unfortunately, for complex systems with a high amount of firewall rules, memory problems arise during the verification process. Based on the complexity of a firewall and the high number of configuration states (package context and system configurations), the size of the verification state spaces grows exponentially and leads to a lack of memory even when using special Uppaal techniques to reduce the size of the input space.

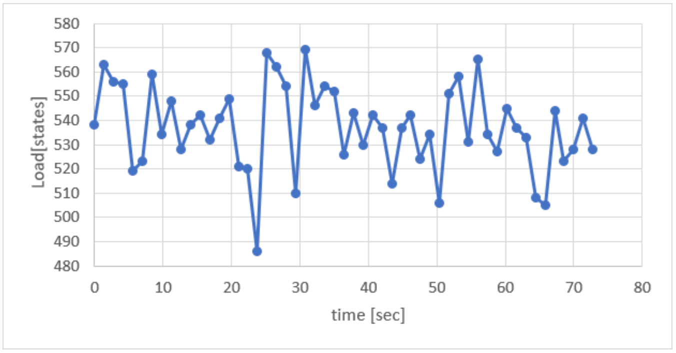


Fig. 6. States Loaded during the Verification

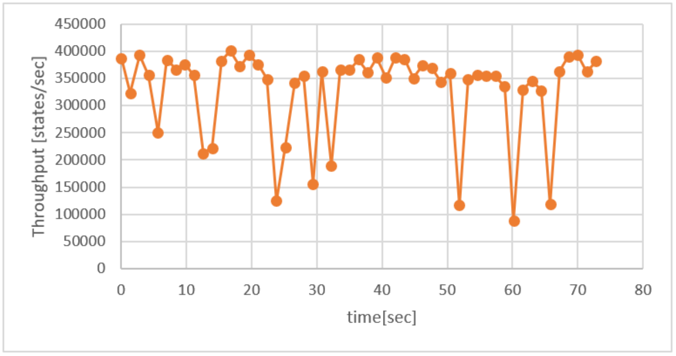


Fig. 7. State Processing Throughput during the Verification

For example, the structure templates in TABLE II will lead to 358.318.080 possible input states[5] for the verification.

In general, model-checkers always face the challenge to efficiently handle the so-called "state explosion problem" [32]. But in case of Uppaal the reason for running out of memory space is the relatively low memory that Uppaal is able to allocate and address. At the moment *"there is no way that Uppaal can address more than 4GB of memory"* [33]. One approach to overcome this issue could be to split a requirement into multiple CTL expressions. For example, if the incoming interface is irrelevant for the requirement, there could be one expression for each interface. This will reduce the state space at the expense of verification time.

VIII. Conclusions and Future Work

The proposed tree based verification method gives the possibility to verify if an implemented firewall satisfies the system requirements with the help of model-checking tools like Uppaal. The given approach cannot only be used to verify the implemented firewall, there is also the possibility to verify the firewall architecture design before implementing it - e.g. a Firewall Decision Diagram (FDD) can be verified. Furthermore, the reversed engineered decision tree can be used to check if the implemented firewall corresponds to the designed one. The proposed methodology in general is also

[5] $358.318.080 = 2^6 \cdot 24^2 \cdot 6^2 \cdot 10^1 \cdot 3^3$

suitable for other rule-based systems and is not restricted to firewalls at all.

We described how to reverse engineer an abstract decision tree from a firewall implementation. The reverse engineered tree enables the simulation of a system's firewall and the verification of the compliance to the identified firewall requirements. Firewall requirements can be translated in CTL expressions and queried against the tree. By running the model-checker in batch mode on a powerful machine and integrating the firewall model-checking process in a DevOps architecture, a large number of firewall requirements can be comfortably verified. In addition, the requirements can be verified independently such that the model-checking process can run on different machines in parallel.

As mentioned above, future activities will investigate the reduction of memory consumption and evaluate the performance of the verification (e.g. time consumption) depending on splitting and limiting the logic of the queries. Furthermore, it is possible to automatically analyze the firewall requirements and generate the formalized CTL queries. The usage of AI/ML (Artificial Intelligence/Machine Learning) techniques for natural language processing (NLP) in this scope is a promising approach that requires investigation in the near future. Finally, there is a need for the extension of the methodology to other components (e.g. routing tables) that address some additional vital security requirements.

References

[1] I. Schieferdecker, J. Grossmann, and M. Schneider, "Model-based security testing," *arXiv preprint arXiv:1202.6118*, 2012.

[2] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Uppaal—a tool suite for automatic verification of real-time systems," in *International hybrid systems workshop*, pp. 232–243, Springer, 1995.

[3] E. Markakis, Y. Nikoloudakis, G. Mastorakis, C. X. Mavromoustakis, E. Pallis, A. Sideris, N. Zotos, J. Antic, A. Cernivec, D. Fejzic, *et al.*, "Acceleration at the edge for supporting smes security: The fortika paradigm," *IEEE Communications Magazine*, vol. 57, no. 2, pp. 41–47, 2019.

[4] J. Corbet, "Nftables: a new packet filtering engine," 2009.

[5] B. Zhang, E. Al-Shaer, R. Jagadeesan, J. Riely, and C. Pitcher, "Specifications of a high-level conflict-free firewall policy language for multi-domain networks," in *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, SACMAT '07, (New York, NY, USA), pp. 185–194, ACM, 2007.

[6] Y. Bartal, A. Mayer, K. Nissim, and A. Wool, "Firmato: A novel firewall management toolkit," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No. 99CB36344)*, pp. 17–31, IEEE, 1999.

[7] S. Pozo, R. Ceballos, and R. M. Gasca, "Afpl, an abstract language model for firewall acls," in *Computational Science and Its Applications – ICCSA 2008* (O. Gervasi, B. Murgante, A. Laganà, D. Taniar, Y. Mun, and M. L. Gavrilova, eds.), (Berlin, Heidelberg), pp. 468–483, Springer Berlin Heidelberg, 2008.

[8] M. G. Gouda and A. X. Liu, "Structured firewall design," *Computer networks*, vol. 51, no. 4, pp. 1106–1120, 2007.

[9] A. X. Liu, "Firewall policy verification and troubleshooting," *Computer Networks*, vol. 53, no. 16, pp. 2800 – 2809, 2009.

[10] A. X. Liu, M. G. Gouda, H. H. Ma, and A. H. Ngu, "Firewall queries," in *Principles of Distributed Systems* (T. Higashino, ed.), (Berlin, Heidelberg), pp. 197–212, Springer Berlin Heidelberg, 2005.

[11] M. Madhuri and K. Rajesh, "Systematic detection and resolution of firewall policy anomalies," *Int. J. Res. Comput. Commun. Technol.(IJRCCT)*, vol. 2, no. 12, pp. 1387–1392, 2013.

[12] E. S. Al-Shaer and H. H. Hamed, "Modeling and management of firewall policies," *IEEE Transactions on network and service management*, vol. 1, no. 1, pp. 2–10, 2004.

[13] K. Karoui, F. B. Ftima, and H. B. Ghezala, "Formal specification, verification and correction of security policies based on the decision tree approach," *International Journal of Data & Network Security*, vol. 3, no. 3, pp. 92–111, 2013.

[14] J. Jürjens and G. Wimmel, "Specification-based testing of firewalls," in *Perspectives of System Informatics* (D. Bjørner, M. Broy, and A. V. Zamulin, eds.), (Berlin, Heidelberg), pp. 308–316, Springer Berlin Heidelberg, 2001.

[15] Adel El-Atawy, K. Ibrahim, H. Hamed, and Ehab Al-Shaer, "Policy segmentation for intelligent firewall testing," in *1st IEEE ICNP Workshop on Secure Network Protocols, 2005. (NPSec).*, pp. 67–72, Nov 2005.

[16] D. Hoffman, D. Prabhakar, and P. Strooper, "Testing iptables," in *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '03, pp. 80–91, IBM Press, 2003.

[17] D. Hoffman and K. Yoo, "Blowtorch: A framework for firewall test automation," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, (New York, NY, USA), pp. 96–103, ACM, 2005.

[18] D. Senn, D. Basin, and G. Caronni, "Firewall conformance testing," in *Testing of Communicating Systems* (F. Khendek and R. Dssouli, eds.), (Berlin, Heidelberg), pp. 226–241, Springer Berlin Heidelberg, 2005.

[19] E. S. Elmallah and M. G. Gouda, "Hardness of firewall analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, pp. 339–349, May 2017.

[20] A. Khoumsi, M. Erradi, and W. Krombi, "A formal basis for the design and analysis of firewall security policies," *Journal of King Saud University - Computer and Information Sciences*, vol. 30, no. 1, pp. 51 – 66, 2018.

[21] C. Diekmann, L. Hupel, J. Michaelis, M. Haslbeck, and G. Carle, "Verified iptables firewall analysis and verification," *Journal of Automated Reasoning*, vol. 61, pp. 191–242, Jun 2018.

[22] R. Khelf and N. Ghoualmi-Zine, "Ipsec/firewall security policy analysis: A survey," in *2018 International Conference on Signal, Image, Vision and their Applications (SIVA)*, pp. 1–7, Nov 2018.

[23] N. Tcholtchev and R. Chaparadza, "Autonomic fault-management and resilience from the perspective of the network operation personnel," in *2010 IEEE Globecom Workshops*, pp. 469–474, Dec 2010.

[24] A. Liakopoulos, A. Zafeiropoulos, C. Marinos, M. Grammatikou, N. Tcholtchev, and P. Gouvas, "Applying distributed monitoring techniques in autonomic networks," in *2010 IEEE Globecom Workshops*, pp. 498–502, Dec 2010.

[25] T. Kastrinogiannis, N. Tcholtchev, A. Prakash, R. Chaparadza, V. Kaldanis, H. Coskun, and S. Papavassiliou, "Addressing stability in future autonomic networking," in *Mobile Networks and Management* (K. Pentikousis, R. Agüero, M. García-Arranz, and S. Papavassiliou, eds.), (Berlin, Heidelberg), pp. 50–61, Springer Berlin Heidelberg, 2011.

[26] R. Chaparadza, M. Wodczak, T. Ben Meriem, P. De Lutiis, N. Tcholtchev, and L. Ciavaglia, "Standardization of resilience amp; survivability, and autonomic fault-management, in evolving and future networks: An ongoing initiative recently launched in etsi," in *2013 9th International Conference on the Design of Reliable Communication Networks (DRCN)*, pp. 331–341, March 2013.

[27] L. N. Tidjon, M. Frappier, and A. Mammar, "Intrusion detection systems: A cross-domain overview," *IEEE Communications Surveys Tutorials*, vol. 21, pp. 3639–3681, Fourthquarter 2019.

[28] M. A. Lopez, A. G. P. Lobato, O. C. M. B. Duarte, and G. Pujolle, "An evaluation of a virtual network function for real-time threat detection using stream processing," pp. 1–5, Feb 2018.

[29] A. Basu, T. Dimitrakos, Y. Nakano, and S. Kiyomoto, "A framework for blockchain-based verification of integrity and authenticity," in *Trust Management XIII* (W. Meng, P. Cofta, C. D. Jensen, and T. Grandison, eds.), (Cham), pp. 196–208, Springer International Publishing, 2019.

[30] O. Andreasson *et al.*, "Iptables tutorial 1.2. 2," *Copyright© 2001–2006 Oskar Andreasson, GNU Free Documentation License*, 2001.

[31] M. Chiari, D. Mandrioli, and M. Pradella, "Temporal logic and model checking for operator precedence languages," *arXiv preprint arXiv:1809.03100*, 2018.

[32] A. Valmari, "The state explosion problem," in *Advanced Course on Petri Nets*, pp. 429–528, Springer, 1996.

[33] "Bug 63 - st9 bad alloc exception." https://bugsy.grid.aau.dk/bugzilla/show_bug.cgi?id=63. Accessed: 2019-08-30.