Middlebox Models in Network Verification Research

Julius Michaelis
Advisor: Cornelius Diekmann
Seminar Innovative Internettechnologien WS2015
Chair for Network Architectures and Services
Fakultät für Informatik, Technische Universität München
Email: michaeli@in.tum.de

ABSTRACT

To address the challenges arising from the development of computer network management over the past decades, researchers have developed a number of tools to assist the operation of networks and help administrators avoid mistakes. These tools often follow the approach to verify an existing network configuration. This poses the problem that the behavior of a lot of potentially complex networking device configration has to be supported. The usual approach to this is to develop simple models that only reflect the aspects of the system that the tool can understand. We survey the related literature for the use of this type of model.

Keywords

Computer Networks, Formal Models, Network Verification

1. INTRODUCTION

Over the past decades, computer networks have grown considerably in size and complexity. Attempts to fulfil the resulting complicated service requirements has given rise to more and more complicated middleboxes (switches, routers, firewalls, etc. . .). Configuring these middlboxes poses an enormous challenge to network operators, who have to be able to understand numerous configuration languages and manage interoperating distributed configuration. While this earned network operators the title "masters of complexity" [24], it is generally seen as problematic.

Computer networks researchers have recognized that other fields, e.g., programming languages, have developed high level approaches to mitigate complexity and present users with simple ways of detecting and avoiding errors. In the past decade, research transferring these approaches to networking has gained traction. A number of tools have been developed that can be invoked to analyze an existing network configuration. The usual approach of these tools is to have the user collect the configuration of his network on a single machine. This configuration is complex and can thus not be directly and likely not fully understood. A tool will parse the configuration and translate it into a representation it can reason about: a model. Depending on the type of the tool, different reasoning is possible; a very common application is to find all possible routing loops — or, if none are found, to prove the absence. However, the development of such tools holds the same challenges that network management holds: a lot of diverse configuration languages and devices has to be supported. *Software Defined Networking* (SDN) attempts to alleviate this burden for operators by proposing a central, programmable controller. This controller is connected to all network devices and can configure them. The devices can notify the controller when they receive a certain type of packet, e.g., packets belonging to a new connection. The controller hands these notifications to a program written by the user. This program can then configure the devices accordingly, e.g., create a path for the new connection. SDN allows to manage nearly all configuration in a single language and logically on a single host.[1] This greatly simplifies the operators tasks and can simplify the verification of the configuration by automated tools.

The crucial steps when attempting to verify a configuration are defining a model of how the configuration is going to be understood, and how to translate real configuration into a representation in the model. This holds various challenges, some of them are intrinsic to modeling: if the model is too simple, it may not be able to represent reality. If it is too complex, it may lose its purpose, as reasoning about it becomes as complicated as reasoning about the original objects. Other challenges are specific to the problem at hand: no tool can possibly understand all the different configuration languages. Typical tools, such as Anteater [18], Hassel [16], VeriCon [5], or Exodus [23] are able to understand subsets of a few configuration languages, such as Cisco IOS or Juniper configurations. This is usually accompanied by the claim that other configuration languages can be easily translated in the same manner, without giving further considerations to the subtleties of such a translation.

In this paper, we survey the related work for the use of models. We focus on the analysis of how systems are modelled.

The rest of this paper is organized into two parts: Section 2 contains the survey of the different models and what aspects they model. We have grouped the models into subsections by the type of the device that is modelled. Section 3 contains a short overview of what purpose the models serve and whether they could be repurposed.

2. BOX MODELS

We continue with the different box models. Section 2.1 surveys link layer switches, Section 2.2 routers, Section 2.3 SDN switches, and Section 2.4 Firewalls. Section 2.5 looks at models of devices that do not only provide a single function. Additionally, Section 2.6 shows the *big switch model*.

[1]Note that this reflects the understanding of SDN from the view of OpenFlow [20], variants exist.

doi: 10.2313/NET-2016-07-1_07

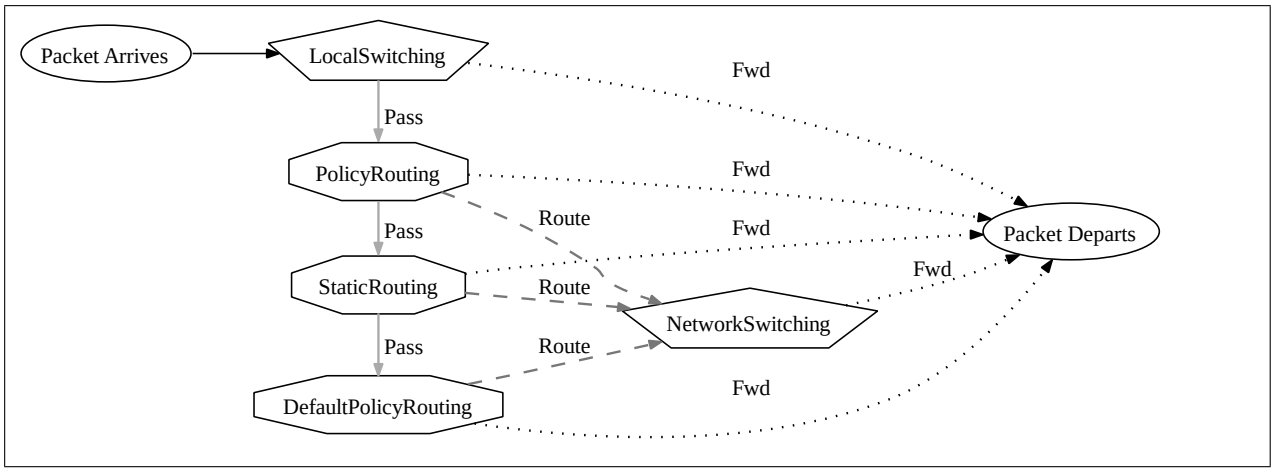

Figure 1: Margrave's router model, taken from [22].

2.1 Link Layer Switches

When considering only the basic switching functionality, Link layer switches become relatively simple devices. They neither offer many opportunities for modeling, nor are they very interesting as a target for verification, since most switches simply have no configuration to verify. Accordingly, there is not much material in the related literature.

There is one problem that arises when verifying networks that contain switches among other devices. Switches are stateful devices, while some verification systems do not support state. A simple modeling solution for that is presented in Header Space Analysis [16]:

> When we generated box transfer functions, we chose not to include learned MAC address of end hosts. This allowed us to unearth problems that can be masked by learned MAC addresses but may surface when learned entries expire.

While this modelling decision has consequences for the models of a variety of devices, it implies that switches are always in their learning phase, i.e. are effectively replaced by broadcast devices.

2.2 Routers

For this section, we will focus solely on layer 3 forwarding.

While Margrave [22] is originally a tool for the analysis of firewalls, it also has a detailed understanding of the packet forwarding process in Cisco IOS to be able to accurately perform its tasks. The model that Margrave uses is shown in Figure 1. The process of forwarding is described as follows. First, packets destined to locally attached subnets are filtered out and directly forwarded. All other packets are subjected to routing. The second step is thus to handle policy routing — packets with special routing rules that do not only depend on the destination address but e.g., on the source address, too. The third step is to consider statically configured routes. All remaining packets are processed using the default policy.

A more abstract model of routing is presented by Xie *et al.* [25].[2] They model a network of routers to be an annotated graph $(V, E, \mathcal{F})$ where the nodes V represent the routers, E contains two directed edges for each physical link, and the edge labels $F_{u,v} \in \mathcal{F}$ express which packets are allowed to flow over an edge u, v. The routing process is modelled using $\mathcal{F}$: a flow over an edge u, v will only be permitted by $F_{u,v}$ if the router u has a route to v for that specific flow.

To summarize, the Margrave's model [22] describes the routing process while the model in [25] abstracts it away to be a property of a graph.

2.3 SDN Switches

This section surveys selected works on switches in software defined networking (SDN). Various approaches to SDN exist. This section focuses on OpenFlow [20] since it is currently the most actively researched variant. We assume that the reader is familiar with its basics. While it could be said that the OpenFlow switch specification [3] itself is based on a model of a generic networking device, we are not going to explore this and instead examine models of OpenFlow switches. We will continue to denote OpenFlow switches as switches in this section for succinctness. The term *datapath element* would be more accurate since the switches can take arbitrary functions.

Guha *et al.* [14] present a fully machine-verified implementation of a compiler for the NetCore controller programming language. For this purpose, they give a detailed model of an OpenFlow switch that adheres closely to version 1.0 of the OpenFlow switch specification [2], including packet processing and switch-controller interaction. We will examine the most important details and begin with the flow table evaluation semantics: Guha *et al.* dedicate significant attention to how a packet is matched against a flow table entry. Their main concern there is related to behavior that was only made explicit in later versions of the specification, e.g. [3, §7.2.3.6]:

[2]The Anteater tool [18] mentioned in Section 1 is based on this work.

$$\frac{\begin{array}{c}\exists(n, pat, \{\!|pt_1 \cdots pt_n|\!\}) \in FT.\\ \quad pk\#pat = \textbf{true}\\ \forall(n', pat', pts') \in FT.\ n' > n \Rightarrow\\ \quad pk\#pat' = \textbf{false}\end{array}}{[\![FT]\!]\ pt\ pk \rightsquigarrow (\{\!|(pt_1)\cdots(pt_n)|\!\}, \{\!|\}\!\})}\ (\textsc{Matched})$$

$$\frac{\forall(n, pat, pts) \in FT \qquad pk\#pat = \textbf{false}}{[\![FT]\!]\ pt\ pk \rightsquigarrow (\{\!|\}\!\}, \{\!|(pt, pk)|\!\})}\ (\textsc{Unmatched})$$

Figure 2: Flow table semantics by Guha *et al.*, taken from [14].

> The presence of an [OpenFlow match] with a given [type] may be restricted based on the presence or values of other [matches], its prerequisites. Matching header fields of a protocol can only be done if the OpenFlow match explicitly matches the corresponding protocol.

For example, to match an outgoing SSH connection, a match must check for at least layer 4 destination port 22, layer 4 protocol TCP, and layer 3 protocol IP. If only the match for layer 4 destination port is included, some implementations of an OpenFlow switch return an error as required by the specification [3, §7.5.4.3]. Others, including the reference implementation [1], silently drop them, which has led to several severe bugs, according to [14]. Guha *et al.* specify their packet matching semantics to only evaluate matches when a previously executed match on the preconditions has assured that the necessary header fields are present.

Next, they specify a flow table to be a multiset of triples of priority, a match condition, and a multiset of output ports. Although a multiset allows for uncountably many flow table entries instead of a bounded number thereof, the implications for the validity of the model are minimal. The semantics $[\![FT]\!]\ pt\ pk \rightsquigarrow (o, c)$ for evaluating such a table is shown in Figure 2. The semantics describes the decision for a flow table FT and a packet pt arriving on a port p. It can specifiy to forward the packets on the port set o or to send the messages c to the controller. The operator $\#$ matches a packet against a rule. Note that this semantics is nondeterministic: if there are multiple matching flow table entries with the same priority, it can be said that all of their actions are executed nondeterministically. This is used to model the fact that the specification [2, §3.4] says that the switch is free to choose any order between overlapping flow entries. For this paper, we have verified that determinism can be enforced by adding the following precondition on the flow table:

$$\forall(n, pat, pts) \in FT.\ \forall\left(n', pat', pts'\right) \in FT \setminus \{(n, pat, pts)\}\,.\\ n = n' \implies \nexists pk.\ pk\#pat \wedge pk\#pat', \tag{1}$$

i.e. for two rules with the same priority, no packet matches both. Note that this is slightly stronger than necessary to make the semantics deterministic: overlapping entries could be shadowed by a rule with higher priority.

Guha *et al.* also specify a semantics for the message processing and passing between switches and controllers. They model it as an inductively defined relation on the states of switches, controller(s) and links between them. The semantics of this is comparatively large: its 12 rules span an entire page. There is one important modelling detail that can be singled out: Switches are modelled as a tuple of their unique identifier, their ports, one flow table, and four message queues, one for each combination of in/out and controller/switch to switch. These message queues are multisets. On the receipt of a message through a link, or when obtaining a message through processing at a switch, the message is first enqueued in one of these queues. The semantics is non-deterministic and allows to accumulate arbitrary many messages and dequeue them in an arbitrary order. This models the option for switches to reorder messages. The only exception is a *BarrierRequest*, which is never enqueued but, given that the input queue is empty, directly processed. It can thus be used to ensure that all messages have been sent before it is processed.

Orthogonal to the work of Guha *et al.* stands VeriCon [5]. It is not a verified compiler for controller programs but a verifying tool for controller programs. It does not have a detailed model of single switches against which it verifies the output of its compiler. Instead, it checks its result on a high-level model of a network of switches. Its authors, Ball *et al.*, begin by presenting a simple example programming language for controllers, called CSDN, and give a formal semantics for this language. VeriCon allows to prove correctness of programs within these semantics but it does not establish the correctness of the compiler. VeriCon takes three inputs: a CSDN program, a topology invariant, and a correctness condition. The topology invariant allows to limit the possible changes in topology, e.g., the user can define that they will always ensure that no path in the network has more than 3 hops. The correctness condition is then verified to hold for all possible states and topology changes. To achieve that, VeriCon uses the following high-level model of a network of switches: Its state is modelled as 5 relations. The first relation contains the links between switches, or switches and hosts, the second one all paths that are possible over those links. These two relations are mainly used to formulate topology invariants. The third relation $S.ft\,(Src \to Dst, I \to O)$ records whether switch S has a rule in its forwarding table to forward packets from host Src to Dst from input port I to output port O. Similar to that $S.send\,(Src \to Dst, I \to O)$ records whether such a packet has actually been sent. Lastly, the relation $S.rcv^{this}\,(Src \to Dst, I)$ models whether a packet has been received at input port I. With these, given a desired postcondition Q, VeriCon can compute the weakest precondition $wp[\![c]\!](Q)$ for executing a command c in CSDN. For example:

$$wp\,[\![pktIn\,(s, p, i) \Rightarrow c]\!]\,(Q) :=\\ \left(s.rcv^{this}\,(p, i) \wedge s.ft\,(p, i \to o)\right) \implies wp[\![c]\!]\,(Q)\,. \tag{2}$$

This is the precondition semantics for the event handler specification $pktIn(s, p, i) \Rightarrow c$. In the event of receiving a packet p at port i of switch s, c is executed. The semantics expresses the following: given that such a packet is actually received and a forwarding rule is installed, the handle has to satisfy the weakest precondition of its command.

Similar to VeriCon is NICE [8]. It uses model checking and other techniques to verify the correctness of a controller program at runtime. It models an SDN as a system of stateful "components" that communicate in a first-in first-out manner. Communication between the components is, among other things, modelled by state transitions in the system. The controller programs are modelled accordingly: as a set of event handlers that trigger state changes in the controller. For model checking, NICE executes these handlers to explore the state space and see if any of the transitions can violate correctness invariants.

NICE notes that, for model checking, it would also need to explore the state space of the switches. Since even the reference implementation Open vSwitch [1] has multiple hundred KB of state when executed, this is not directly feasible. NICE thus presents a simple model of a switch with a reduced amount of state. A switch is modelled as a set of communicating channels, two state transitions and a single flow table. Except for the control channel, which operates strictly in a first-in first-out manner, these channels may also drop or reorder messages[3]. On the receipt of at least one message, a state transition is executed. To reduce the amount of state transitions necessary, all packets present in a state are modelled as being processed as a single transition. NICE also makes an important remark on the flow table: two flow tables can be syntactically different, i.e. have a different entry structure, but be semantically equivalent, i.e. lead to the same forwarding decisions. This observation is true for all three models here. For example, a table that contains only exact flow matches (flow entries without any wildcards) makes decisions independent of the priority of the rules (i.e. the order in which they are considered)[4]. NICE uses heuristics to merge semantically equivalent states.

2.4 Firewalls

The term firewall is used for a diverse variety of devices and software. Devices by different vendors, such as Cisco, Sun Microsystems, or Sophos have a largely different set of features and purposes. Even the Linux kernel has two different firewall implementations (iptables and nftables). This means that a large number of different models exists. Nevertheless, a common principle can be factored out: most firewalls and all models considered here consist of rules, which in turn consist of at least a match and an action. The match decides whether the action is to be applied to a given packet. The firewalls' types differ in how rules are organized, i.e. in which order they are applied, and what kind of match expressions and actions are supported. Another detail of interest is how connection state tracking is modelled, i.e. how packets that belong to established connections are treated differently from packets for new connections. Many real world firewalls begin by accepting packets that belong to or are related to an established connection. Finding a simple but powerful model for state is hence important.

Accompanying a model of a firewall, there always has to be a model of packets on which the firewall operates, albeit this is often left implicit. One of the few works that explicitly specifies the packet model is [6]:

$$(\alpha, \beta)\, packet \coloneqq$$
$$(id \times protocol \times \alpha\, src \times \alpha\, dest \times \beta\, content)\,. \qquad (3)$$

This can be read as: given arbitrary types α and β, a packet consists of a record of a unique identifier, the used protocol (http, ftp, ...), a source and destination address of type α and packet content of type β.[5] It is obvious that this format does not model real packets very closely, since neither the used (application layer) protocol is usually stated directly, nor is every connection associated with a unique ID. Nevertheless, the ID hints to how state is modelled by Brucker et Wolff in [6]. They model state by allowing the match to consider a list of all packets that the firewall has accepted so far. The ID can be used to determine if a packet is the first of its connection.

A less complicated model of state, which is also based on the packet format, can be found in ITVal [19] (however, not in a strongly formal manner). Whether a packet is part of an established connection is simply treated to be another packet field. The theory files accompanying [12] contain a proof that that model is not weaker than querying an internal state table when performing a stateful match.

The packet model is often tightly tied to which types of match expressions the firewall supports. A very common subset that can be found in many real firewalls and models is to support equality matches on (OSI) layer 4 protocol, source, destination ("ports"), the physical ingress port and additionally prefix matches on the layer 3 addresses. Some models extend this by fields for TCP flags [19, 26], or connection state [6, 22].

Besides the set of supported match expressions, firewalls and models also differ in how these expressions can be combined and how these combinations are represented. Margrave [22] supports conjunctions of disjunctions, i.e. it allows to specify several possible values for one field and allows combining fields while requiring all of them to match. *Iptables Semantics* [12] by Diekmann *et al.* allows for more complicated expressions: given *match* is a match on a single field, it supports the following match expressions *mexpr*:

$$mexpr \coloneqq match \mid \neg mexpr \mid mexpr \wedge mexpr \mid \texttt{True} \qquad (4)$$

This model is a superset of what iptables supports: iptables supports negation of matches only on the lowest level, i.e. it only supports constructing $\neg match$ but not $\neg mexpr$. This is an example (but not the only one) of a model that could have been easily made to mirror a system more closely but instead was made more powerful. In this case, *mexpr* allows to express arbitrary boolean functions, which can be used to compute the expression for packets that are not matched by a rule.

Packet and match models that are tailored to be suitable for the implementation of an analysis can be found in FIREMAN [26] and ITVal [19]. FIREMAN models a packet as a

[3] Note the difference to [14] where the control channel does also not operate in a first-in first-out manner and can reorder messages.

[4] Assuming that the switch does not accept overlapping rules.

[5] Brucker et Wolff later specify α to be a four-tuple of integers to represent the IP-address in dotted-decimal notation and a port, also represented by an integer (i.e. a number from $\mathbb{Z}$).

doi: 10.2313/NET-2016-07-1_07

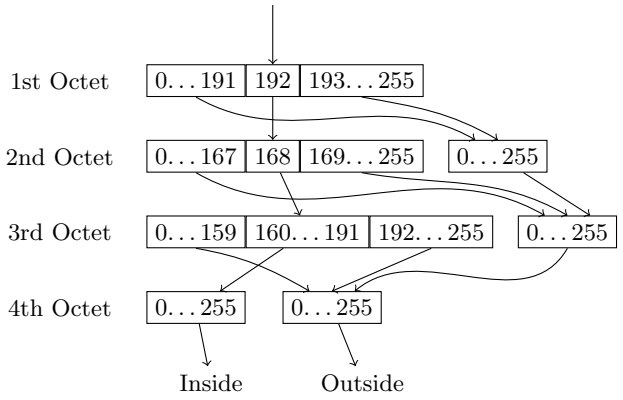

Figure 3: Example of how ITVal [19] would represent a match for 192.168.160.0/19, given that the packet consists only of a single IP address to match on.

vector of bits that represent its header. Match expressions are boolean expressions that can be efficiently represented by *Reduced Ordered Binary Decision Diagrams (ROBDD)* [7]. ITVal [19] extends this to *MDDs* [17], a structure that is similar to a *ROBDD* but allows continuous values for its variables. Consequently, ITVal models packets as the vector of bytes that represent source and destination for IP address and layer 4 port. Additionally, it keeps separate fields for the layer 4 protocol type, the TCP flags and the connection states. Each byte and field is then represented by one level in the MDD. Figure 3 shows an example of how an MDD is used to match a single IP prefix. The purpose of this subdivision of the packet header is to create a balance between too many levels and too much information on a single level of the MDD.

After considering the match of a rule, the action has to be modelled. The common subset of actions that can be found in all models we analyzed is to either let packets pass the firewall or to stop them. Iptables supports a number of other actions that are directly executed, such as `LOG`, which will generate debug output but have no effect on forwarding, or `REJECT`, which will stop the packet and additionally send an error message. The semantics by Diekmann *et al.* [12] shows a way to translate action types with behavior unkown to the system back to only forwarding or discarding the packet. The model of actions given by Brucker *et* Wolff [6], allows for something more complicated: the action returns a packet. This allows to model packet modification by firewall rules.

The last important property of a firewall model is how rules are combined to form the firewall. Most models can be categorized to either use what Yuan *et al.* [26] call the *simple list model* (used e.g. in [22]) and the *complex chain model* (used e.g. in [19]). The list model states that the firewall rules are written as one list that is traversed linearly. Each rule either applies and the execution terminates or the execution continues with the next rule. The chain model extends this by allowing for multiple lists and the possibility to conditionally jump to the start of such a list and to conditionally return to the origin of the jump. Diekmann *et al.* [12] formalized both and present a translation from the chain model to the list model.

If a firewall is an actual networking device, it also needs to decide on which port to forward packets, i.e. become a switch or router additionally to its firewall function. This type of combined functionality is considered in the next section.

2.5 Complex devices

Real network devices often fulfil more than one of the functions described above. A common example of this is a Cisco IOS router, which is usually configured with both an ACL (i.e. its firewall function) and routing information.

An important insight is that these functions usually have very little or no relevant shared state. The key implication of this is that the different stages of such a system can be analyzed separately and then *pipelined* together. In the example of the IOS router, this would mean to first analyze the incoming ACL, then the routing configuration and then the outgoing ACL.

Dobrescu and Argyraki [13] have realized that this holds true even for controller software that is written for SDN switches. When attempting verification of software in general, one has to deal with the path explosion problem. By dividing a network system into m independent elements with maximally n branches each, pipelined analysis can reduce the amount of paths that has to be analyzed exponentially from $\mathcal{O}(2^{mn})$ to $\mathcal{O}(m2^n)$. Combined with further optimization for relevant data structures and symbolic computation, their tool *ClickVerifier* is able to verify controller programs. Dobrescu and Argyraki make an explicit point of using the pipeline model only to explain why their system has the desired performance. For the verification, the actual generated controller program bytecode is passed to the analysis engine S^2E [10] to avoid any abstraction errors that might happen when modeling OpenFlow switches.

Another interesting instance of the pipelining model can be found in the tool Margrave by Nelson *et al.* [22]. A schematic representation of how it is used to decompose a Cisco IOS configuration can be found in Figure 4. Each element of the pipeline is used to provide the further elements of the pipeline with necessary information to continue the analysis, e.g., the *Internal Routing* step is used to decide which outbound NAT and ACLs apply.

A very similar approach to this is taken in Hassel, the tool that implements Header Space Analysis by Kazemian *et al.* [16]. Hassel translates dumped Cisco IOS configurations into functions represented in a model on which symbolic computation is possible. Representing the full function of a router with a single function of this model would result in very large representations. The transfer through one Cisco IOS router is thus modelled as traversing three layers in the model, one input ACL and VLAN untagging step, one routing step, and one output processing step. The difference to how Margrave uses the pipeline model is that Header Space Analysis reuses the exact same model for each step.

This solution of size reduction is also applied by Exodus [23]. It translates Cisco IOS configurations into controller config-

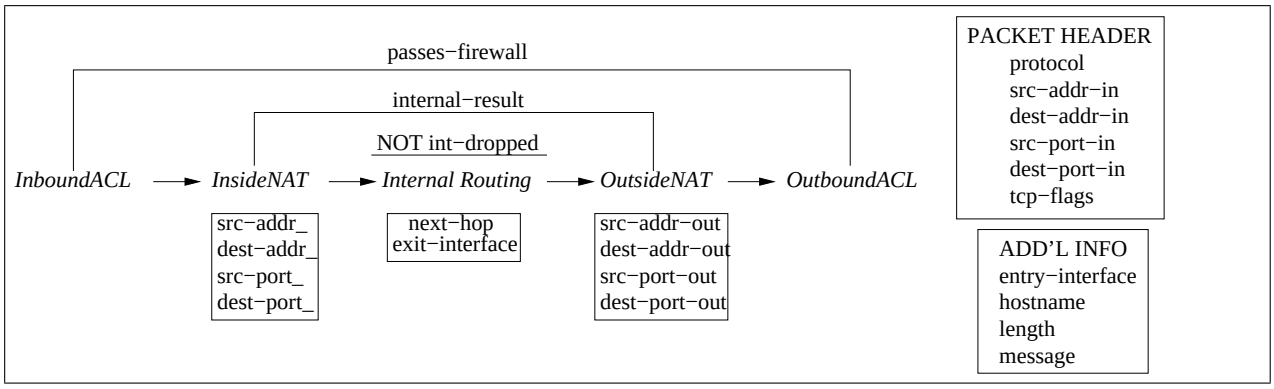

Figure 4: Margrave's decomposition of IOS configurations, taken from [22]

uration for an OpenFlow switch (*cf.* Section 2.3). Much of the available OpenFlow capable hardware only supports a single table of matches and output ports. Similar to the functions of Hassel, compressing the entire functionality of a Cisco switch into a single table would create very large representations through the use of cross products. Exodus thus uses multiple switches (i.e. physically multple devices). Their pipeline steps are, in order, VLAN untagging, input ACLs, routing, NAT, routing (2), layer 2 rewriting, output ACLs, and VLAN tagging.

2.6 Big Switch Model

While the *big switch model* is not strictly speaking a model of a networking device but a model of the entire network, we feel that it is important enough to mention it here. It is used in various places, first and foremost in SDN [9, 15, 21] programming languages, but also e.g., for network verification [4]. The general idea is that a network or subnetwork of switches, routers, firewalls displays a forwarding behavior to all attached devices that are not part of the subnetwork. The subnetwork usually has a complicated distributed configuration that defines its forwarding behavior. Even in SDN programming languages, this state is often exposed to the programmer. Proponents of the big switch model usually attempt to represent this state as if it was the configuration of a single switch with each of its ports representing one connection from the subnetwork to something outside of it. The rationale for this is that a representation for the configuration of the big switch could be smaller and thus easier to understand or verify.

Anderson *et al.* [4] propose a slightly different interpretation of the big switch model: for a network to be an instance of the big switch model, they require that the network displays the behavior of a big learning switch, i.e. implements all-pairs reachability. Since NetKAT [4] allows testing the equality of two network descriptions, they formulate a formal condition for this to be true. Showing this condition here would require explaining the formalism used by NetKAT and is thus out of scope.

3. COMPARISONS

In the previous section, we surveyed for existing models and what they express. This section gives an overview of what type of model they are, i.e. how they are used and integrated in their environment. The results of this section are summarized in Table 1. For the table, we differentiated between two types of model usage: are the models used in a *proof* that establishes correctness properties of the system, or are they used when *implementing* a surrounding system. Additionally, we checked if some kind of formalization was available and ready for reuse. Some of the attributions are not entirely clear. We will explain them in the following paragraphs.

Work		Proof	Implementation	Reusable
NetKAT	[4]	✔	✔	✔
VeriCon	[5]	✔	✔	✘
HOL-TestGen	[6]	✔	✔	✔
NICE	[8]	✘	✔	✘
Iptables Semantics	[12]	✔	✔	✔
(Dobrescu and Argyraki)	[13]	✘	✘	✘
(Guha *et al.*)	[14]	✔	✔	✔
HSA / Hassel	[16]	✘	✔	✘
ITVal	[19]	✘	✔	✘
Margrave	[22]	✘	✔	✔
Exodus	[23]	✘	✔	✘
(Xie *et al.*)	[25]	✘	✘	✔
FIREMAN	[26]	✘	✔	✘

Table 1: Usage of models in the surveyed work

Most of the models considered here fall into one of two usage categories. Members of the first category [8, 16, 19, 22, 23, 26] propose a model of networks or networking devices that simplifies reality significantly. This model is then used to justify and explain the steps that have been taken in the implementation of an accompanying tool. The level of formality of these models varies. Also, the models are usually tightly tied to the implementation and not suitable for reuse in other projects. Margrave is a notable exception to this as its model is a relatively generic representation of Cisco IOS configuration.

Members of the second category [6, 12, 14] give a model that has a high degree of formality and is accompanied by a semantics that aims to closely mirror the behavior of the real system. This semantics of all of the models from the second category is available in form of code for theorem proving software.

Some models could not be sorted into either of these categories.

- VeriCon [5] explicitly formalizes the model it uses and includes a semantics. However, its semantics does not mirror the behavior of any real device. Nevertheless, the use of the *Satisfiable Modulo Theories* solver Z3 [11] does provide proof that the programs checked with VeriCon are indeed correct wrt. the semantics. VeriCon's authors also claimed that the semantics would be made available online. To this date, it is marked as "pending".
- Similar to VeriCon's case is NetKAT [4], except the formalization is available.
- While Dobrescu and Argyraki [13] do propose a model, they do not use it in their implementation. This is, as mentioned, to avoid carrying any discrepancies between the model and reality into the implementation. The model is merely used to justify why the implementation can terminate quickly.
- The work by Xie *et al.* [25] gives a model of a network of routers but does not implement anything based on it. An implementation that is based on that network model has later been given with Anteater [18] by different authors. As such, the model by Xie *et al.* proved to be (re-)usable even though there is no readily available formalization of it other than the publication itself.

4. CONCLUSION

We surveyed related work for the use of models of networking boxes. We included models of learning switches, routers, OpenFlow switches, firewalls, and devices that include multiple of these functions. This work can provide a reference for further works in the area that want to use strong formalism and thus have to use models of networking boxes. It can answer the questions of which models have already been constructed, how they are used, and how their properties make them qualified for the specific use-case.

5. REFERENCES

[1] Open vSwitch. `http://openvswitch.org/`.

[2] OpenFlow Switch Specification v1.0.0, December 2009.

[3] OpenFlow Switch Specification v1.5.1, March 2015.

[4] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. *ACM SIGPLAN Notices*, 49(1):113–126, 2014.

[5] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. VeriCon: Towards verifying controller programs in software-defined networks. In *ACM SIGPLAN Notices*, volume 49, pages 282–293. ACM, 2014.

[6] A. D. Brucker and B. Wolff. Test-Sequence Generation with HOL-TestGen with an Application to Firewall Testing. In *Tests and Proofs*, pages 149–168. Springer, 2007.

[7] R. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8):677–691, August 1986.

[8] M. Canini, D. Venzano, P. Peresini, D. Kostic, J. Rexford, et al. A NICE way to test OpenFlow applications. In *NSDI*, volume 12, pages 127–140, 2012.

[9] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the Network Forwarding Plane. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, pages 8:1–8:6, New York, NY, USA, 2010. ACM.

[10] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*. Citeseer, 2009.

[11] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.

[12] C. Diekmann, L. Hupel, and G. Carle. Semantics-Preserving Simplification of Real-World Firewall Rule Sets. In N. Bjørner and F. de Boer, editors, *FM 2015: Formal Methods*, volume 9109 of *Lecture Notes in Computer Science*, pages 195–212. Springer International Publishing, 2015.

[13] M. Dobrescu and K. Argyraki. Software dataplane verification. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI), Seattle, WA*, 2014.

[14] A. Guha, M. Reitblatt, and N. Foster. Machine-verified Network Controllers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 483–494, New York, NY, USA, 2013. ACM.

[15] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "One Big Switch" Abstraction in Software-defined Networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 13–24, New York, NY, USA, 2013. ACM.

[16] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, 2012. USENIX.

[17] H.-T. Liaw and C.-S. Lin. On the OBDD-representation of general Boolean functions. *IEEE Transactions on computers*, (6):661–664, 1992.

[18] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review*, 41(4):290–301, 2011.

[19] R. M. Marmorstein and P. Kearns. A Tool for Automated iptables Firewall Analysis. In *Usenix*

annual technical conference, Freenix Track, pages 71–81, 2005.

[20] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[21] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker, et al. Composing Software Defined Networks. In *Networked Systems Design and Implementation*, pages 1–13, 2013.

[22] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave Tool for Firewall Analysis. In *Proceedings of the Large Installation System Administration Conference*, 2010.

[23] T. Nelson, A. D. Ferguson, D. Yu, R. Fonseca, and S. Krishnamurthi. Exodus: toward automatic migration of enterprise network configurations to SDNs. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 13. ACM, 2015.

[24] S. Shenker, M. Casado, T. Koponen, and N. McKeown. The future of networking, and the past of protocols. Talk at Open Networking Summit, 2011.

[25] G. Xie, J. Zhan, D. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of IP networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 2170–2183 vol. 3, March 2005.

[26] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra. Fireman: a toolkit for firewall modeling and analysis. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15 pp.–213, May 2006.